GRAPHICAL INTERFACE BETWEEN THE CIRSSE TESTBED AND CIMSTATION SOFTWARE WITH MCS/CTOS

by

Anna B. Hron

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering Department
Troy, New York 12180-3590

December 1992

CIRSSE REPORT #131

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I would very much like to thank my advisor Dr. Steve Murphy for all his support and guidance throughout this project, and for his insistence, from the start, that the ideas were plausible. Without his endless patience and encouragement during all phases of my work, this thesis would have never been completed. I wish him much success in his new position.

I would also like to thank Ms. Bonnie Kortyna for her advice, support and friendship, without which this effort would have seemed impossible. There are a few people at the lab whose expertise, insight and patience did not go unnoticed: thanks to Eric Ameres and Jon Weaver as well as everyone at CIRSSE, and also to Dr. Steve Derby for his support. Special thanks to Keith Nicewarner for sharing his knowledge of programming tirelessly, and allowing me to incorporate his socket routines into my research. Also, I am forever in debt to my parents, who have supported me emotionally and financially throughout my education and my life, and who *always* believe in me. Lastly, I would like to thank Mr. Rodney Hylton for being my motivation to complete this work on schedule.

ABSTRACT

This research is concerned with developing a graphical simulation of the testbed at the Center for Intelligent Robotic Systems for Space Exploration (CIRSSE) and the interface which allows for communication between the two. Such an interface is useful in telerobotic operations, and as a functional interaction tool for testbed users. Creating a simulated model of a real world system, generates inevitable calibration discrepancies between them. This thesis gives a brief overview of the work done to date in the area of workcell representation and communication, describes the development of the CIRSSE interface, and gives a direction for future work in the area of system calibration. The CimStation software used for development of this interface, is a highly versatile robotic workcell simulation package which has been programmed for this application with a scale graphical model of the testbed, and supporting interface menu code. A need for this tool has been identified for the reasons of path previewing, as a window on teleoperation and for calibration of simulated vs. real world models. The interface allows information (ie. joint angles) generated by CimStation to be sent as motion goal positions to the testbed robots. An option of the interface has been established, such that joint angle information generated by supporting testbed algorithms (ie. TG, collision avoidance) can be piped through CimStation as a visual preview of the path.

CHAPTER 1

INTRODUCTION

The Center for Intelligent Robotic Systems in Space Exploration (CIRSSE) was established by the National Aeronautics and Space Administration (NASA) in 1988 as part of a national program to integrate university research with their own, and contribute to the rapidly growing field of robotics. The main emphasis of the center is on intelligent machines, with technical support in the areas of sensing, control, real-time computing and the contribution to autonomous and telerobotic systems [2]. The research described in this thesis applies to the goals of the center, in that a graphical interface with the manipulators and their environment is of critical importance for support of telerobotic operations.

1.1 Goals and Motivation

The goal of this work was to develop a convenient and efficient interface between a user of the CIRSSE testbed and the hardware and software of the testbed itself. Prior to this work, all interaction with the testbed was handled via the CIRSSE testbed Operating System (CTOS) (section 2.3.1) or other off-line programming techniques. No teach pendant, nor any other convenient method was available to implement simple, routine tasks on the manipulators. This design, using the CimStation graphical package as an interface, provides not only a convenient tool to handle standard testbed operations, but is an exact graphical replica of the robots and their environment, which allows the user to execute involved paths and preview their performance. The ability to have a window on the motion of robotic manipulators is critical in a teleoperational mode, and for verifying the accuracy of a predetermined path.

The motivation for this research was to facilitate the interaction between the user and the CIRSSE testbed, by implementing a front end graphics package which requires only a limited knowledge of CAD type software, to engage the testbed arms for demonstration, research, or task oriented purposes. This application extends to uses in space or any other teleoperational situation, where the user is not able to see the arms directly or by way of camera images, and relies fully on a graphical representation of the world as her/his window on the environment. In addition, this interface has the capability of accepting information (ie. path definitions) from external routines and allows a screening process of the motion without risking damage to the equipment in the event of a unforeseen path error.

1.2 Historical Review

In any research or industrial setting where an operator is forced to be at a location removed from the robotic workcell, there is a need for teleoperational capabilities. By definition, a teleoperable system [3,23] involves a human operator controlling a manipulator from a position which is not necessarily within visual proximity. With the number of sites engaging in robotic activity constantly growing, so is the amount of work being done in the area of workcell representation and telerobotic operation. In this section, a concise review of some of that work will be covered, and a comparison made with the research of this thesis. Future work in the area of calibration, as it applies to the topics of this thesis, together with possible approaches, will be covered in Chapter 5.

Many sites have invested time into developing a world model of their workcell environment, for off-line programming and teleoperational purposes. In the case of telerobotic manipulation, the need for a world model is clear, but often, a simulated model of any kind, is beneficial for proof of concept development or testing. The University of

California (Davis, CA) [21], has created a CAD-based programming and simulation system, which accepts both sensor input and off-line command to represent their multiple arm environment . The kinematic and physically described system updates the image of primitive shapes with every acquisition of new information. The University of Science and Technology in Wuhan, China [8], has developed an off-line, computer aided motion package, for use with an IBM-PC AT computer to operate a PT-300V robot. The user employs a menu interface for direct communication with the manipulator, and the results can be represented by a kinematically and dynamically accurate three dimensional simulation. Both motion planning and collision detection capabilities are included. Park and Sheridan [20], implement an IRIS workstation as the as the user's interface to a system which supports both a manual and supervisory mode of teleoperation. The operator generates instructions interactively, and heuristic algorithms return targets which are attained without collision. This work is based heavily on the use of sensory feedback. All of the above products are similar in nature to the CIRSSE effort in world modeling, but are not as concentrated on the graphical aspects.

Off-line programming involves creating a series of robot motions which will accomplish a desired task taking into account physical constraints, before the robot is engaged, and there have been various tools developed to simplify this process. Smith of Hewlett Packard Laboratories in Palo Alto, CA [28], has developed a higher order robot interface, in which the user supplies less rigorous, abstract commands to advance the robot through its task. Mazer, et alii [17] use a classical simulator and graphics for off-line programming, and a crude graphics scheme to communicate control commands through and ethernet system. In the area of telerobotics, Tendick et alii of the University of California (Berkeley,CA) [30], have developed a feed forward, vision based control system for their vision calibration capabilities. One of the most important features of the CIRSSE testbed interface, is the off-line programming function which allow the user to completely define and test a motion sequence prior to downloading it to the manipulators. In addition, effective on-line options are available for interactive manipulator control.

Perhaps the research which is most comparable to that of this thesis, is the work being done by the Jet Propulsion Laboratories (JPL) in California [9,10,13]. They have developed a state of the art robotic facility as part of the NASA telerobotics program, which includes support for autonomous motion, dual arm force reflecting teleoperation with voice interaction, and shared control for autonomy. Their world model is calibrated with the physical workcell at runtime upon obtaining sensor data. This is replaced with a more involved routine if the information of several sensors is to be processed and cross referenced. The off-line programming modes include: task level, process level and servo level in decreasing complexity respectively. Compared to the capabilities of the CIRSSE testbed, those of the JPL testbed are far more advanced in the area of control and sensory feedback. The advantage again, of the CimStation interface, is its robust flexibility in the area of graphical representation and manipulation.

1.3 Research Organization

As with any user interface, it is desirable for the internal structure of the algorithm to be completely invisible. In this application, the execution was envisioned to be from a single "machine" which would oversee both the display of CimStation, as well as the interface which translates the requests sent from the graphics session, into commands which operate the testbed hardware. The current software which exists to support the CTOS, and that which supports CimStation are not compatible. CTOS runs on VxWorks and CimStation utilizes SunView. A direct communication between these two systems has not been established to date, therefore the UNIX environment was implemented as a liaison for the graphics application under SunView, and the message handling task under VxWorks.

This research was separated into four phases which progressed into the current version of the graphical interface. In the first phase, a detailed model was constructed with the tools provided by CimStation for that purpose. Though a kinematic model of the PUMA 560 exists as a feature of CimStation, both left and right CIRSSE end effectors had to be duplicated graphically, as well as the three degree of freedom K.N. Aronson platforms (section 3.1). With this complete, the first pass for communication between CimStation and the testbed controlled by CTOS was attempted successfully, by using data files (section 3.2). Information being generated by functions within CimStation (internal and formulated) was sent to and saved by a data file accessible by both processes. Consequently the message handling routine running under CTOS, polled the data file at a

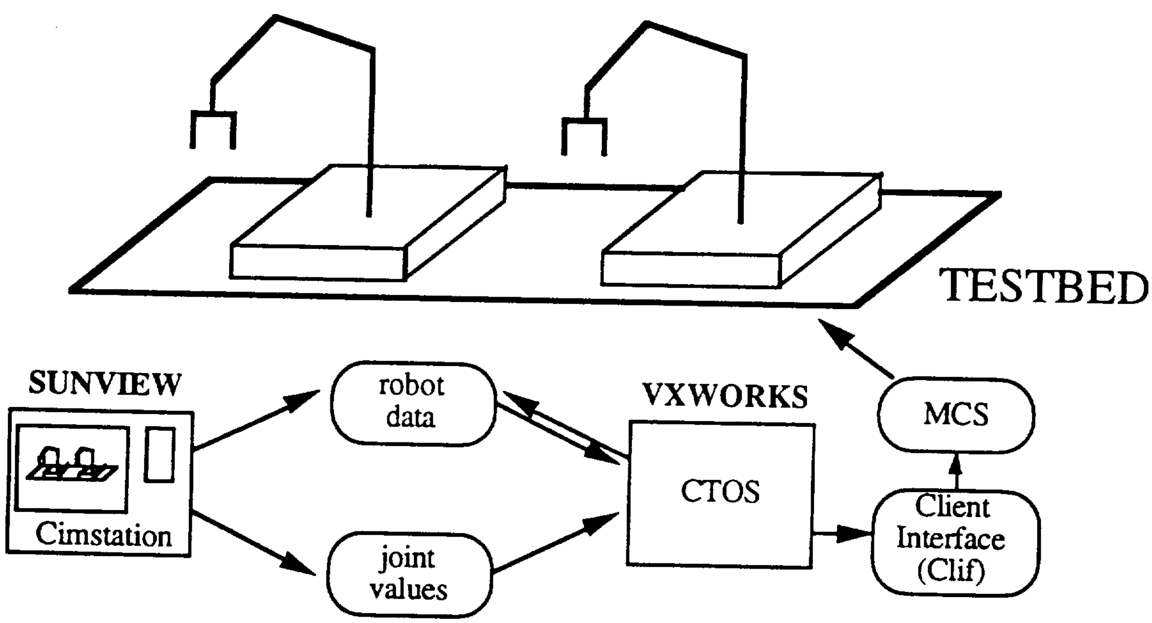


Figure 1.1 - Message Passing Scheme of the Graphics Interface

repeated interval of 1 second for any new information, interpreted the meaning and sent the appropriate command to the Motion Control System (MCS). A layout of this exchange is shown in Figure 1.1. With this complete and functioning, the same scheme was achieved by substituting a "socket" structure for the data files (Chapter 4).

1.4 Design Constraints

The constraints which were identified in each phase of research are described here.

• Message Passing by Data Files - to ensure accessibility of both processes to the message files, a rigorous scheme of opening and closing the files before and after each read/write operation was maintained, and for this reason, the number of files, and the information passed to them was kept small.

• Message Passing Through Sockets - the extremely involved and time consuming procedure by which external C code, is incorporated into CimStation, necessitated the minimization of corrections and code changes. Also, the extraction of information from CimStation proved to be non-trivial, and therefore, data was prepared within CimStation, to facilitate its post-processing in the supporting CTOS routines.

The amount of data, and the work involved in pre/post-processing it, determines the efficiency (ie. speed) of the interface, and therefore, in both phases of research, improved efficiency was obtained by minimizing the quantities of each.

1.5 Advantages and Uses

In addition to having a high quality graphical display of the robotic environment being studied, CimStation has capabilities which allow it to be a useful tool in other areas of testbed research. Several areas which are currently under development in CIRSSE have found the CimStation package, and the subsequent graphical interface, to be a valuable tool. For example, the Geometric State Manager (GSM), developed as a world model of the testbed, with a graphics display as one of its features, imports the models created with the CimStation modeling package. The graphical representations are saved as IGES (Initial Graphics Exchange Standard) files, and are easily transported to the GSM for use in its representation of the Testbed environment. IGES is a standard being developed by the

ANSI Y14 committee to transfer primarily CAD data as a conglomerate of geometric primitive shapes such as points, lines, cubes, cylinders, etc.[1]. Some other areas of research which have implemented the graphical interface, are the single/dual arm collision detection algorithm and the trajectory generator (TGen). Both use the package as a front end graphical previewer for their generated paths as a visual check for path validity in addition to their numerical results.

As a stand alone routine, the graphical interface between CimStation and the CIRSSE Testbed serves many purposes in addition to the ones described above. Currently, its usefulness will be seen mostly in the CIRSSE PUMA robot laboratory, where it will be employed as a straightforward tool for robot manipulation and path planning. CimStation has the advantage of doing "visual" motion, that is to say motion which requires a visual estimate of the end effector's position in space, as opposed to a rigorous evaluation of the joint angles. This feature allows the user to program a general path, and view it without having to engage the robots until the path is satisfactory. The interface, when communicating, sends successive joint angles of an engaged robot, in real-time to the testbed robots. In addition, this interface has potential applications in many telerobotic situations such as hazardous materials operations where it is unsafe for a human to operate, or in a space robotic situation where the routine duties are often performed by robots with astronaut teleoperation to reduce the risk of human injury. In both cases, an accurate window on the environment is required for adequate performance by the manipulator.

1.6 Interface Software

The supporting software for the final design of the graphical interface, is located on the CIRSSE computer network, in the directory

/home/hron/interface

This directory contains all the source code for the CimStation user menu (written in SIL), and the source code for the CTOS task (written in C). A listing of these codes may be found in Appendix C.

CHAPTER 2

DESCRIPTION OF RESOURCES

The resources employed in the course of this research include the CIRSSE Testbed hardware (ie. the robots, end effectors and platforms), the testbed software, CTOS and the CimStation graphics package which is the area of emphasis, and the computer facilities as a link between them. In this chapter, each of these resources will be discussed in detail, to provide a background understanding of the interface to the reader.

2.1 CimStation Software

This software, developed by Silma, Inc. [25,27], is highly versatile graphical, robotics, simulation package, which allows the user to implement any number and variety of environments. The features and capabilities of CimStation are particularly suited for this application, as the CIRSSE testbed consists of customized equipment which had to be modeled specifically. The package allows the user to interact with predefined robots and workcells via user-friendly menu functions, or create a more personalized session by writing code in the Silma language SIL [26]. The flexibility of this software, provided this research with the necessary tools to implement a detailed model of the testbed, and the supporting code to fit the specific needs of the Center.

2.1.1 Graphics

CimStation is a three dimensional modeling package which represents all of its objects as groups of primitive shapes and solids such as blocks, cylinders, prisms etc. These models may be defined as robots, end effectors or simply objects with which the other elements of the workcell interact. Any type of model may be created within the session or a CAD model may be imported via the IGES (Initial Graphics Exchange Standard) representation. Conversely, models and representations of robots may also be exported from CimStation, which is a feature that was used frequently to supply other testbed applications such as the CIRSSE Geometric State Manager (GSM) with precise descriptions of the testbed.

Internally, all solids are built and stored using the boundary representation method, and have the option of being displayed as solid models with hidden lines removed, or as wireframe models to conserve memory and thereby increase display efficiency. This structure organization allows solids to be grouped (permanently or temporarily) into meaningful workcells within which the robot(s) operate. Due to this representation, the models are strictly kinematic entities which only know physical properties such as gravity, collision with other solids and dynamics if intentionally supplied with that supporting information. Background code and customized software packages can be created or installed from an outside source. A dynamics package, and a collision detection package are currently available from Silma, as well as group operations, coordinated motion, painting and external devices packages. On/off-line translating packages are available from Silma, which essentially perform the same function as the interface component of this research, but the CimStation translator for the PUMA robot controllers is written to manipulate VAL II commands, however the CIRSSE controllers do not use VAL II. In addition to the incompatability of programming languages, the interface to multiple PUMAs is not adequate, the translator can not handle more than six DOF and is not programmed with iinformation about the CIRSSE grippers. Therefore it was determined that this package would not be of use in this research.

The CimStation environment consists of a full screen window which is dedicated to displaying the graphics portion of the software, and the menu interfaces. A background window runs as a display for all SIL output, messages and errors. This window can be disregarded by the novice user, but proves to be critical in the development of SIL code. Figure 2.2 shows a general

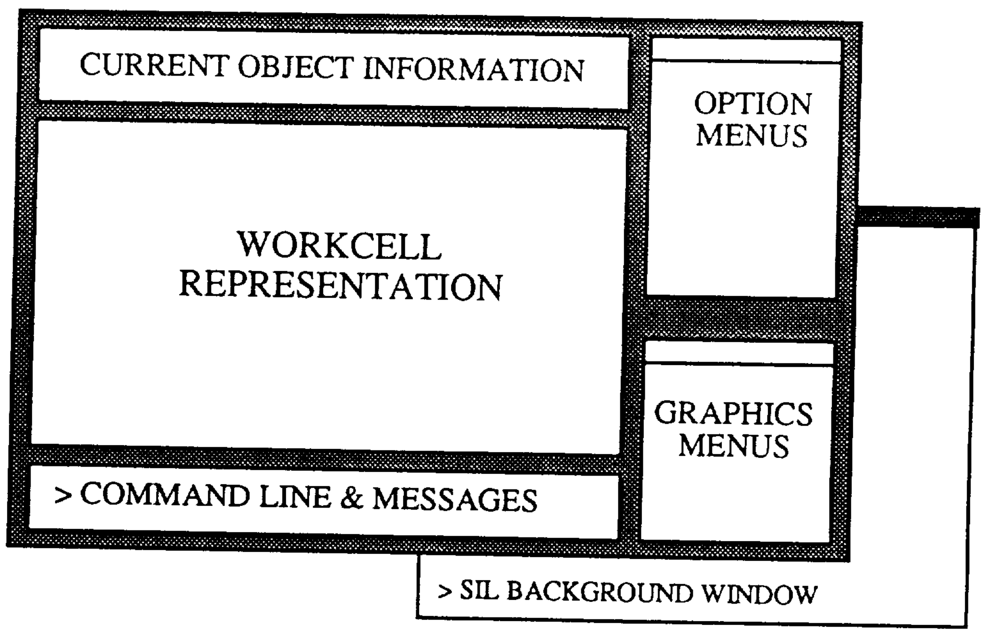


Figure 2.1 CimStation Session Windows

configuration of the CimStation session. The *current object information* subwindow contains data concerning the cycle time of an operation, the current robot/object being operated on and in the case of a robot, its current joint angles for all its n joints (these are updated with the completion of each move). The *command line* subwindow contains a SIL prompt, and allows the user to interact directly with the internal language of the software, or any code which has been written for that particular application. The *option* and *graphics* subwindows allow full operation of CimStation through menu driven commands. Any command which can be executed with a menu choice, has a counterpart which can be typed on the SIL command line. Finally, the *workcell representation* window shows the current state of the manipulators and their environment. This is perhaps the most impressive feature of CimStation, in that it is so versatile in the variety of ways it allows the user to view the workspace; from any of the world orthogonal directions (top, front & side views), and from any spin or tilt direction. The combination of these views allows for a complete set of configurations for the user to choose from.

The two subwindows in the CimStation session window which are menus, are the options menus, and the graphics menus (Figure 2.2). All the functions for constructing workcells, robots and end effectors are contained in the options window, along with the commands to move objects (both in world coordinates and in the case of robots via a teach

CIMSTATION
MODELING
LAYOUT WORKCELL
PROGRAMMING
RUN SIMULATION
GENERATE OUTPUT
APPLICATIONS

<table>
<tr><th colspan="6">UTILITIES</th></tr>
<tr><td colspan="2">REF F</td><td colspan="2">LIBR</td><td colspan="2">HIDE</td></tr>
<tr><td colspan="3">TO TOP</td><td colspan="3">STATUS</td></tr>
<tr><td colspan="3">GRAPHICS</td><td colspan="3">GROUPS</td></tr>
<tr><td colspan="3">COLLISIONS</td><td colspan="3">TRACING</td></tr>
<tr><td colspan="3">UNDO MOVE</td><td colspan="3">MEASURE</td></tr>
<tr><td colspan="3">DYNAMICS</td><td colspan="3">SHOW MOVIE</td></tr>
<tr><td colspan="3">PAUSE:</td><td colspan="3">off/on</td></tr>
<tr><td colspan="3">PRESET TASKS</td><td colspan="3">APPLICATION</td></tr>
<tr><td colspan="3">TO TEXT</td><td colspan="3">FILE SYSTEM</td></tr>
</table>

Figure 2.2 - CimStation Session Menus

pendant), program the robots and create animations/movies. The graphics menus give the user control over what is being displayed such as a toggle on the various reference frames attached to each object, or even the object itself. Also, choices can be made as to how the objects are displayed; as wireframe models, with hidden lines removed or as regular surface solids, and at what rate the screen should refresh itself.

2.1.2 Motion and Kinematics

These features are what make CimStation so effective as a modeling tool. The metakinematics package is the environment which supports the creation and modification of robots and end effectors. The information which must be supplied by the designer is limited to number and size of links (fingers), and their relation to each other, and the various joint limits which should be imposed on the robot. A knowledge of the Denavit-Hartenberg [4] coordinate frame labeling is helpful in specifying exact orientations of the links, but is not required for successful manipulator and end effector construction. The limits imposed on the formulation of a robot include a minimum number of links (three), and that they be an open kinematic chain (ie. link i must necessarily be connected to link i+1) proceeding from base to end effector. This became a concern in this application, as the entire testbed is an 18 degree of freedom manipulator, and consists of two end effectors which CimStation is not able to support. This topic and its solution will be discussed in Chapter 3. In the case of gripper construction, CimStation limits the number of "fingers", but is very flexible in defining their operation; the open and close positions can be specified, as well as the speed with which they grasp and the percentage of full "open" each operation is to include.

The motion of any robot can be defined in several ways under CimStation. For general or individual motion, the user can select a mode under the move menu which specifies what plane (X-Y, Y-Z, X-Z) or what axis of rotation (X, Y, Z) the move will take place on/about. Then either a numerical answer may be typed in at the command line, or a graphics pick may be made with the mouse to define the exact location/orientation of the manipulator and its end effector. This provides the option to make moves visually, or with exact numerical information. For path motion, after each move has been completed, it is possible to attach a "frame" of reference to that point (location in space of the end effector), with respect to the current governing reference frame, which can be anywhere, but usually is at world zero. These frames can then be combined in some order to represent the via points of the desired path. Once a path has been defined, it can be stored along with its environment, and any of the additional features mentioned in Section 2.1.4 can be implemented.

The graphics and manipulation features of CimStation are compatible to any generic CAD package with options such as copy, rotate, translate and also the full range of solid modeling objects to draw on. Once the robots, objects and end effectors have been constructed, CimStation includes convenient options for moving entire groups of objects or only parts. In the case of end effectors, individual menu options exist for the manipulation of gripper functions (ie. as described above). The robots can be positioned with respect to the environment (world) or moved in a kinematic sense with the robot pendant feature, by a vector of joints, or in terms of world position and orientation.

2.1.3 SIL Programming Language

In addition to the menu functions which exist in a CimStation session, it is also possible to interact with the software via SIL [26] commands. This is the computer language which has been developed by Silma Inc., to support the graphics package. Any command which has a corresponding menu button, can be translated into one or a series of SIL commands to perform the same task. SIL is a Pascal-based language, and as with Pascal, is highly modular. SIL is defined using the LISP (LISt Processing) language [7] which is commonly used in AI (artificial intelligence) research, but is used here for its exclusive use of lists for data and program structures. At compilation time, all SIL programs are translated into C [12] before execution, and for this reason,CimStation will allow external C code (eg. the testbed interface) to be integrated in. This is a critical component of the second phase of this thesis, the interface of CimStation with MCS/CTOS using sockets.

2.1.4 Additional Features

In addition to the basic features which have already been discussed, CimStation offers many others which deserve mention here. Several of these features were applied in the development of the graphical interface, and as it is possible that these and others could be implemented in future improvements of the interface. In the area of path planning; currently the Trajectory Generator (TGen) developed at CIRSSE, is being used to guide the physical arms from one joint vector position to another, whereas CimStation contains an accurate path planner which is used to guide the graphical arms. The CimStation path planner accounts for joint angle and work envelope limits, and does not permit even the kinematic arms from attempting to reach an unobtainable goal position. Also accelerations and velocities are taken into account when defining a path, and just as with the testbed controller, these influence the trajectories. Either straight line or joint interpolated motion can be chosen as governing processes. Another feature which is available, but not taken advantage of, under the version 4.2.1 of CimStation, is the dynamics package. A PID controller is used to model and simulate the effects of inertia and friction, and can be defined for each joint individually. For the interface on this work, only the kinematics of CimStation were employed, and other information such as dynamics and other path planning information was taken from the software developed specifically for the testbed.

Two features which are not available in the version 4.2.1, are the collision-free path planning option and the robot calibration option . The former involves defining a path as a succession of Cartesian points (or of joint angle vectors), and the software generating a path free of collisions with any other objects in the work space. This is a critical option in any workcell programming environment, especially when a front end graphical display is not available, and the motion of the physical robot is the test for collision!..This is a feature which will be available in future versions of CimStation (ie. version 4.3.1 [24]), and so for this interface, the collision detection will be done with the algorithm developed at CIRSSE for that purpose. Another critical component of any physical environment simulation, is calibration of the two. The difficulty of synchronizing the modeled workcell with the physical one is sizeable. Version 4.3.1 contains a calibration package which takes into account both the accuracy of the robot and in the placement of the workcell parts. The need for workcell calibration will be discussed further in Chapter 5. More information about 4.2.1 and subsequent versions of CimStation and software features can be found in [27].

A feature which is used to implement externally generated data, is the concurrent programming option. It is possible, through SIL *processes* (similar to functions and procedures), to manipulate more than one robot simultaneously. This compensates for CimStation's inability to command more than one robot at one time in the on-line mode, but off-line, with a SIL program loaded, any number of manipulators can be in motion, on individual paths, concurrently.

2.2 Hardware

The testbed is composed of two, six degree of freedom (DOF), Unimation, PUMA 560 manipulators, which are each mounted on a three DOF, K.N. Aronson platform [33] and are fitted with pneumatic grippers [29] especially designed for the research being done at CIRSSE (Figure 3.2). These together give the testbed a total of 18 DOF, with each robot operable separately for performing localized tasks, or all 18 joints together for tasks which require a bigger range of motion. Figure 2.3 shows a representation of the testbed including the strut rack with a strut, taken from the CimStation hidden line removal, wireframe option ([33] contains more information about the specific link frames associated with each DOF). The custom built platform consists of two carts which each have a rotate and tilt joint, and can translate along one linear joint which has a travel range of ±1.5 m along the world Y-axis (Figure 3.2). The global origin, (world) zero frame is shown in

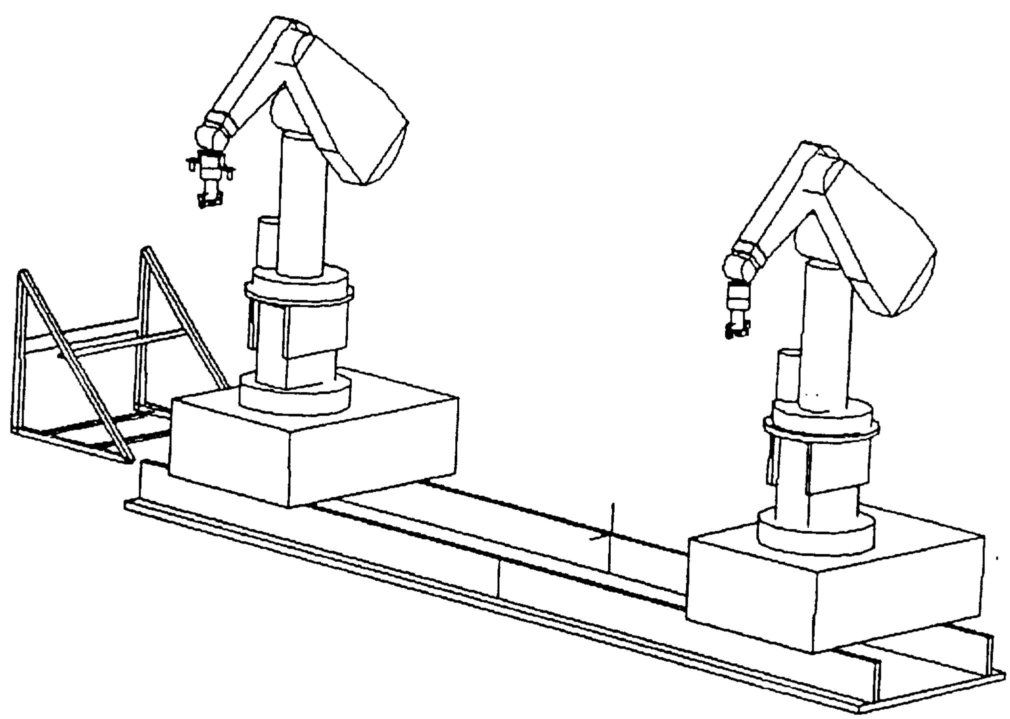


Figure 2.3 - CimStation Representation of the 18 DOF CIRSSE Testbed

Figure 2.3, located on top of the back platform rail, in the middle of its length. The PUMA 560's are mounted similarly to the platforms (as opposed to mirror image of each other), and all robots in the testbed are referenced in terms of their location as shown in Figure 2.4.

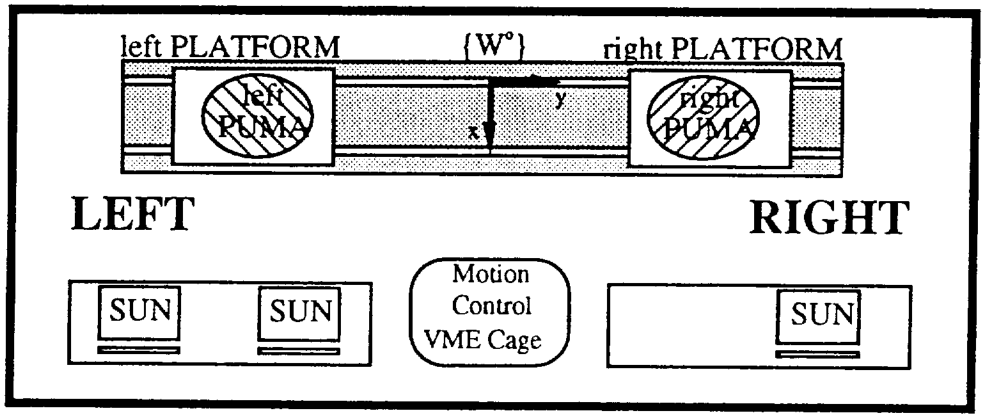


Figure 2.4 - Testbed Labeling and Layout

The 18 joints of the testbed are labeled starting with the first (linear) joint of the left platform, and ending with the last (sixth) joint of the right PUMA, which is labeled joint 18 as shown in Figure 2.3. These joint frames were labeled using a modified Denavit-Hartenberg notation, which is described fully in [4]. This modified notation was used for more convenient controller calculations. Each joint has a mechanical limit in addition to a software the software limit is chosen intentionally within that of the mechanical, in order to prevent any hardware damage due to software motion commands which exceed the physical limitations of the testbed. Joint and link information for the testbed is shown in Table 2.1 [33].

Frame	Name	Global Label	Local Label	Physical Limit	Software Limit
0	World zero	G_0	L_0, R_0	N/A	N/A
1	L Cart linear	G_1	L_1	(-1.372, 0.610) m	(-1.372, 0.610) m
2	L Cart rotate	G_2	L_2	(-150, 150) degs	(-150, 150) degs
3	L Cart tilt	G_3	L_3	(-45, 45) degs	(-45, 45) degs
4	L PUMA shoulder	G_4	L_4	(-256, 79) degs	(-251, 74) degs
5	L PUMA upper-arm	G_5	L_5	(-221*, 40*) degs	(-215, 34) degs
6	L PUMA fore-arm	G_6	L_6	(-60, 246) degs	(-55, 241) degs
7	L PUMA wrist	G_7	L_7	(-126, 150*) degs	(-121, 144) degs
8	L PUMA flange tilt	G_8	L_8	(-100, 100) degs	(-95, 95) degs
9	L PUMA flange rotate	G_9	L_9	(-290*, 290*) degs	(-284, 284) degs
10	R Cart linear	G_{10}	R_1	(-0.610, 1.372) m	(-0.610, 1.372) m
11	R Cart rotate	G_{11}	R_2	(-150, 150) degs	(-150, 150) degs
12	R Cart tilt	G_{12}	R_3	(-45, 45) degs	(-45, 45) degs
13	R PUMA shoulder	G_{13}	R_4	(-253, 83) degs	(-248, 78) degs
14	R PUMA upper-arm	G_{14}	R_5	(-221*, 43*) degs	(-215, 37) degs
15	R PUMA fore-arm	G_{15}	R_6	(-60, 243) degs	(-55, 238) degs
16	R PUMA wrist	G_{16}	R_7	(-134, 153*) degs	(-129, 148) degs
17	R PUMA flange tilt	G_{17}	R_8	(-100, 100) degs	(-95, 95) degs
18	R PUMA flange rotate	G_{18}	R_9	(-290*, 290*) degs	(-284, 284) degs

L - left
R - right

* - not the mechanical limit

TABLE 2.1 - CIRSSE Testbed Joint Coordinate Frames and Limits

The CIRSSE computer facilities [5] consist of eleven Sun workstations (SUN Microsystems, Mountain View, CA), the VMEbus cage and the two VT320 terminals which are connected to the cage. There are a total of five Sun 4 workstations, one of which is in the testbed laboratory, and six Sun 3 workstations. The VMEbus cage currently has six CPU's (Vx0 - Vx5), each of which is accessible through one of the VT320 terminals dedicated to the cage only, and collectively the cage is connected to the rest of the computer Ethernet network through a gateway on CPU Vx0.

2.3 Software

2.3.1 CIRSSE Testbed Operating System

CTOS [14,15,16,19] is the operating system developed over several years by CIRSSE to manage the communication of various tasks, such as those which send software commands to the testbed hardware (MCS), potentially running on several processors or workstations simultaneously, in a real-time, efficient manner. CTOS is written in the C programming language, and therefore any process which hopes to employ it must also be written in C. This was one of the difficulties discovered in interfacing the CimStation software with CTOS processes, because CimStation is written primarily in SIL, which is a Pascal based language (section 2.1.3). The capabilities of CTOS are substantial, and are employed in all research being done on the testbed.

Figure 2.5 shows a skeleton CTOS application, which shows a configuration file (.cfg) delegating tasks and chassis information across the computing network. Each application file

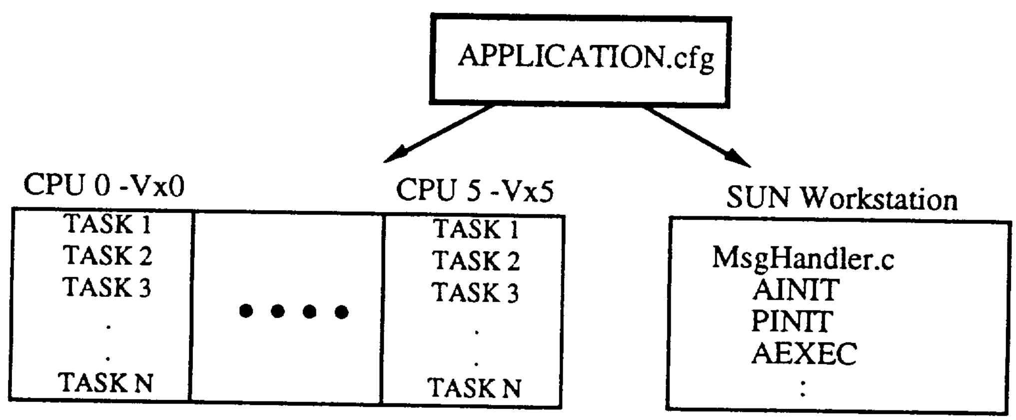


Figure 2.5 - Generic CTOS Application Layout

consists of specifications for which *chassis* (CPU or workstation) the application(s) should run on, the *task* (Message Handler) name(s) and their respective chasses, and a chassis where the I/O operations will be executed. An application file can consist of several configuration files, which can each have a number of tasks they delegate, and all tasks can run on a different chasses, and often do, to distribute computational effort. The tasks consists of the supporting C code which carry out the desired procedures, and define the building blocks of the CTOS application. As described, a CTOS application is versatile enough to be very complex, or simple, this being the strength of such an operating system.

2.3.2 Client Interface

The CLIF (CLient InterFace) is a C function library which can be called to support any other CTOS task.. The CLIF is designed to take motion control (MCS) functions, and combine them into several routines which provide a simple interface for the user who does not want to explore the complexity of MCS commands. The arguments to the various CLIF functions vary, but in general the information which needs to be supplied is:

ARM_TYPE - left 9-DOF, right 9-DOF or all 18-DOF
ROBOT_TYPE - PUMA, platform or both
ROBOT_KEY - an integer key to "reserve" the physical robot

With this information, plus other function specific parameters, the CLIF can command the testbed hardware to engage, determine the current value for specific joints, operate the end effectors, (dis)engage compliant mode and with additional speed and goal information, the robots can be commanded to move. This code is a relatively new area of research within CIRSSE, and the work of this thesis, is the first to implement the CLIF in a non testing capacity.

CHAPTER 3

INTERFACE BY DATA FILES

The first design for developing a graphical interface between CimStation and the CIRSSE Testbed was one for information to be passed between processes through data files and subsequently through the CIRSSE Testbed Operating System . This was a relatively simple concept, as data file Input/Output is routine in both CimStation (SunView) and CTOS (VxWorks). The first step was to develop accurate CimStation models of every object within the testbed workcell. Next, SIL code was written to support user commands from CimStation, and finally, the corresponding C code, in the form of a CTOS message handler was written to accept and interpret CimStation generated information.

3.1 Development of Models

Getting a precise model of the equipment in the testbed, represented within CimStation was the first task of this project. Models were created for the testbed grippers (left, right and original), the testbed platforms (left and right identical), the combination of PUMA and platform (constituting the left and right 9 degrees of freedom) and the strut rack and struts (see Appendix B) for wireframe models). Within CimStation, there exist kinematic models of several commonly utilized industrial robots, such as the PUMA 560 which is the robotic arm used in the CIRSSE testbed, therefore, this is a component for which it was not necessary to create a model. CimStation's modeling capabilities are very comprehensive, and allow a wide variety of objects, such as functioning end effectors and n degree of freedom robots to be constructed. Each model can be specified, within the constraints imposed by the types of geometric shapes available, to look exactly like the real world object.

The CIRSSE testbed employs two grippers [29], one for each PUMA 560 arm (Figure 3.1). The basic shape of each gripper is the same, but one has a set of wrist cameras, and the other does not. There is also the original gripper, which is identical to the others, but without the added features. This is a spare used for making tests and improvements which will later be implemented on the mounted grippers. The end effector for the left PUMA is designated as Gripper A, and the one for the right PUMA is designated as Gripper O. Each has a Lord Industrial Automation Force/Torque sensor mounted between it and the wrist flange of the robot, and the maximum capacities are 15 lbs force, and 50 in-lbs Torque. Gripper O has an additional camera mounting, which consists of an aluminium fixture supporting two small cameras. These end effectors are pneumatically operated, made primarily of aluminum, have boolean crossfire sensors in the

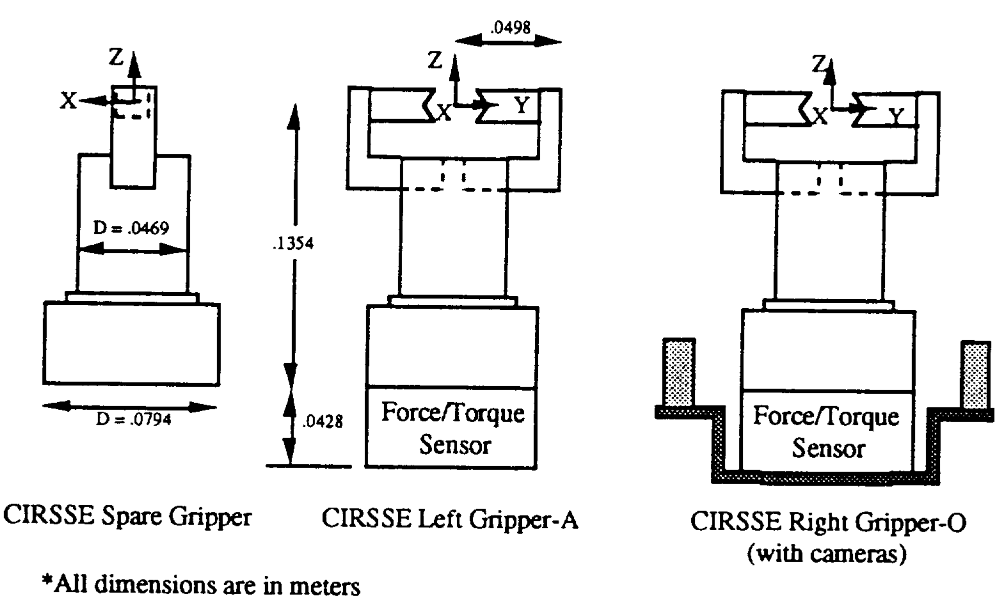


Figure 3.1 - The CIRSSE Testbed Grippers

fingers to detect the presence of an object, and were custom built for easy manipulation of the scaled struts used for research in the CIRSSE testbed.

The two PUMA 560, six degree of freedom industrial robotic arms in the CIRSSE testbed, are each mounted atop a three degree of freedom transporter platform made by K.N. Aronson Co. of Arcade, NY (Figure 3.2). The first joint is linear and allows the eight joints above it to make large translational motions. Joints two and three are rotational pitch and yaw joints, which in combination with the large translational prismatic joint increase the PUMAs' workspace considerably. The PUMAs are secured onto the platforms, and their combined joints are numbered sequentially to produce a nine degree of freedom robotic manipulator, and together with the other PUMA/Platform combination define an 18 degree of freedom testbed, capable of single arm or dual arm functionality.

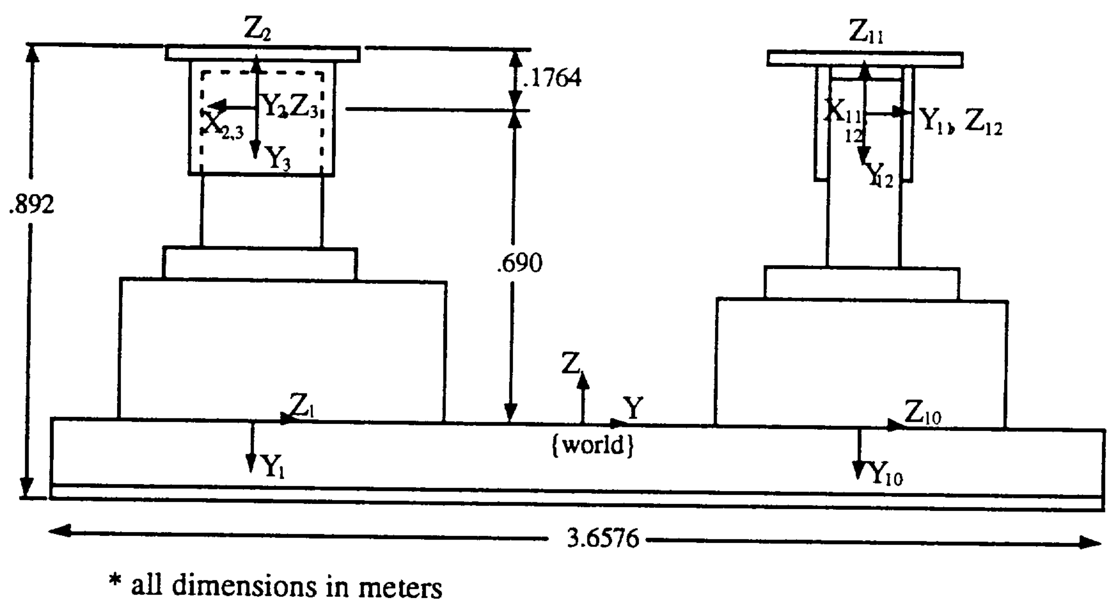


Figure 3.2 - The K.N. Aronson 3-DOF Platforms

Both halves of the testbed were modeled as single 9-DOF robots because the CimStation motion capabilites are limited to actuating one single robot at a time. The physical testbed however is able to engage and operate all 18 degrees of freedom simultaneously. As mentioned earlier, the CimStation software is not able to support a closed kinematic chain, of which the complete CIRSSE testbed is an example, and so therefore that inconvenience was superficially avoided by composing the testbed out of two, nine DOF robots (joints 1-9 and joints 10-18) or out of four robots, two six DOF PUMAs (joints 4-9 and 13-18) and two, three DOF platforms (joints 1-3 and 10-12). Unfortunately, unless CimStation is being fed joint information from some external process (ie. the collision avoidance routine), then it is not possible to represent motion of all 18 DOF simultaneously. The user identifies the type of motion which she will be undertaking, and selects at the start which of the two types of workcells should be loaded.

The CimStation routines are robust in representing non-redundant manipulators (ie. less than or equal to six DOF), however, any number of DOF beyond that, the user is exposed to the possibility that a closed form solution may not exist for that particular configuration. In these cases, CimStation resorts to an iterative solver technique to derive the joint angles for the specified goal. As was seen in the nine DOF PUMA/Platform robot, the iterative solver was employed for nearly every move. A side effect of this iterative solution, in the case of the redundant (9-DOF) manipulator was encountered The linear joint of the nine DOF sequence, has a much larger range of motion than do any of the revolute joints. A large linear motion , which is often desirable, was assigned a proportional penalty, and the iterative solver was reluctant to allow the large prismatic motions of the first joint. The first joint could be operated separately, that is given a specific destination, with accurate results. Only when the iterative solver was engaged was the motion stifled. As was discovered from contacting Silma, Inc. this is a known bug in the software which hopes to be eliminated in future versions.

As CIRSSE is dedicated to research for space applications, much of the work being done there involves the manipulation of scaled struts which will be used as the basic building components of future space structures such as the Space Station Freedom. These building blocks and their repository, the strut rack, were therefore modeled in CimStation for use in interaction workcells of the manipulators and struts (Figure 3.3).

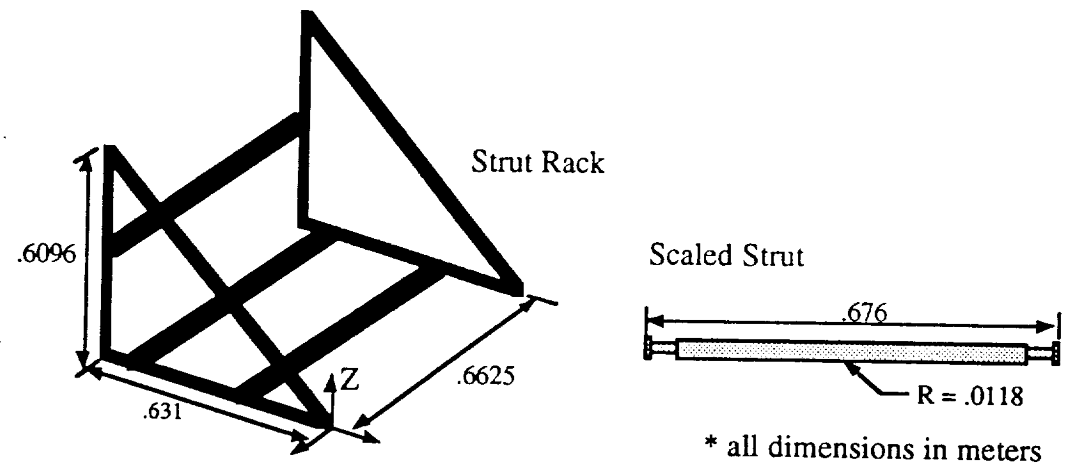


Figure 3.3 - Scaled Strut and Strut Rack

3.2 Logistics of Message Passing

It was determined that the CTOS message handling routines could not directly communicate with the CimStation software, because of an incompatibility in operating environments (ie. VxWorks vs. SunView). Ordinarily CTOS could *spawn* a task which would oversee another process, including the operation of a software package, but they would have to be running in the same environment. So, since both environments are based out of the UNIX operating system, this commonality was used as a connection which could readily be accessed by both processes.

The scheme which was used to pass information from CimStation to the CTOS message handler is shown in Figure 3.4. The commands for action were initiated by the CimStation user. These commands included choices of testbed type, which arm to engage for motion, when to begin transmitting joint information, and when transmission was complete. By utilizing the "user-defined menu" option [27] in CimStation, a customized menu was developed for specific use in the testbed interface. Each command for action discussed above, was actuated with a choice out of the custom user menu. Each command was identified with a string of characters, for example "MOVE" for motion initiation, and this was sent as string data to and saved by the command data file. On the CTOS side, the portion of code which would accept the CimStation information, was given the path name to the intermediary data file, and polled the file for a "new" command every 1.0 seconds. If a new string was detected, then the subsequent lines were read, their values saved under the appropriate variables and upon completion the "old" command was sent so that the current information would not be re-processed.

If the first line of the command file contained the MOVE command, then this indicated that the CimStation user had selected the ON mode, meaning that joint angles were being transmitted to the joint angle file. The CTOS message handler would then read the joint angle file (at a rate comparable to that at which values were being supplied by CimStation) until such time as the command "FINISHED" was encountered. This alerted the message handler that joint angles were no longer being transmitted, and that a resumption of polling the command data file was desired. This is just a concise representation of the scope of the CimStation, UNIX , CTOS Message Handler intercommunication, and a more detailed explanation will be given in sections 3.3 and 3.4.

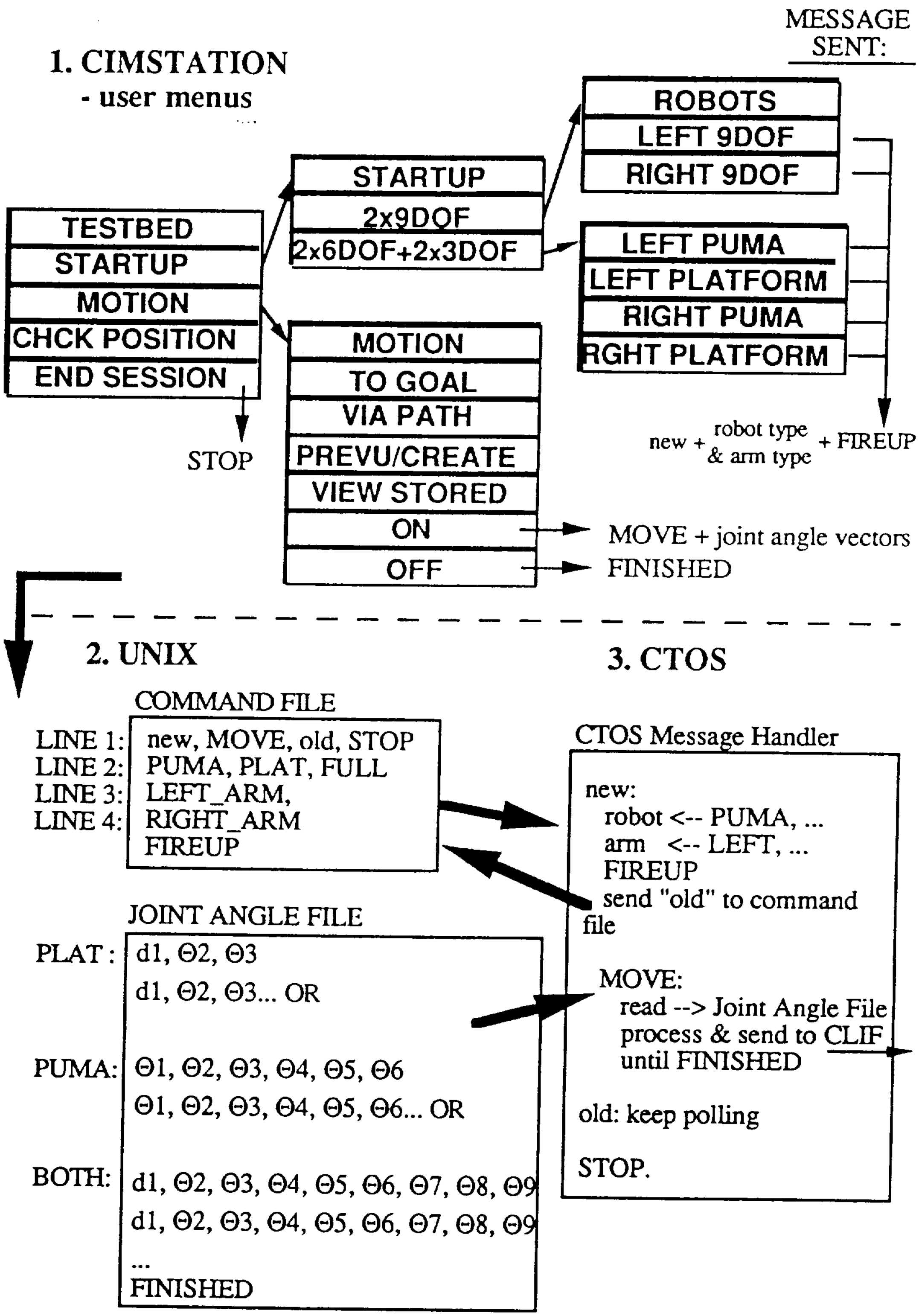


Figure 3.4 - CimStation/CTOS Message Passing Scheme

3.3 CimStation User Menu

In this section, the main user defined menu TESTBED and all its submenus will be discussed in detail. The testbed menu was made accessible (after successful compilation of the supporting code) from the main CimStation menu. The four selections under the testbed menu controlled the initial set up of the workcell, the engagement of motion and the ultimate interruption of communication with CTOS.

3.3.1 STARTUP Menu

The first step the user was asked to take following the initiation of the user menu, was to choose the "type" of testbed which she would be using. The type refers to which form the robots would be loaded into the workcell. As mentioned above, the PUMA graphic models were included with the software, and a kinematic robot representation was created for the 3-DOF platforms. A need was identified for the two to be modeled as one 9-DOF robot, as the capability of controlling one half of the testbed (9-DOF) clearly exists in the physical environment. For this reason, a choice between the loading of two 9-DOF robots (joints 1-9 and 10-18 as individual robots) or two 6-DOF PUMA manipulators plus two 3-DOF platforms to represent the make-up of the graphics testbed was required. The drawback to this method of representation is that the user must know ahead of time in what combinations she will be operating the robots.

Upon selection of testbed type (2x9 DOF or 2x6DOF + 2x3 DOF) the user menu code loaded the appropriate graphics robots in the correct orientation and CIRSSE defined ready position (Table 3.1) onto the screen, and prompted the user to select which robot

LEFT				RIGHT			
PLAT	ready	PUMA	ready	PLAT	ready	PUMA	ready
d_1	−1.3 m	Θ_4	0 °	d_{10}	1.3 m	Θ_{13}	0 °
Θ_2	0 °	Θ_5	−45 °	Θ_{11}	0 °	Θ_{14}	−45 °
Θ_3	0 °	Θ_6	180 °	Θ_{12}	0 °	Θ_{15}	180 °
		Θ_7	0 °			Θ_{16}	0 °
		Θ_8	45 °			Θ_{17}	45 °
		Θ_9	90 °			Θ_{18}	90 °

Table 3.1 - CIRSSE Ready Positions

should be engaged. Depending on which type of testbed was selected initially, a corresponding menu of robots was activated:

<u>2x9 DOF</u>	<u>2x6 DOF + 2x3DOF</u>
LEFT 9DOF	LEFT PUMA
RIGHT 9 DOF	LEFT PLATFORM
	RIGHT PUMA
	RIGHT PLATFORM

With the selection of any of these robots from the TESTBED/STARTUP/ROBOTS menu, the series of strings: new↵

PUMA, PLAT or FULL ↵

LEFT_ARM, RIGHT_ARM↵

FIREUP ↵

was sent to the command file, and saved. The first line denoted that the user made a new selection of a robot, and that the next three lines of the data file should be read and interpreted. Lines two and three contained robot type and arm type information respectively, there being two possible combinations for the 2x9DOF testbed, and four different combinations for the 2x6DOF + 2x3DOF testbed representing the number of individual robots in each workcell type. Line four was intended to be a trigger to another portion of the message handler, and therefore was included in each robot type selection.

Next, the user menu returned to the main TESTBED menu and gave the user the option of motion, or ending the session. From the appearance of the menu, it would have seemed that selecting another testbed and consequently engaging another robot for motion was a feasible choice, however, due to the fact that this first design for the communication of CimStation and CTOS was meant to serve as a proof of concept example, the option to switch between robots was not included. This option was however incorporated into the second design of the interface, as described in Chapter 4.

3.3.2 Motion Menu

In the case of the TESTBED/MOTION option selection, the MOTION submenu was presented with several more choices for the user:

TO GOAL & VIA PATH The alternative of which trajectory generator would control the testbed motion was included, so that the user would have more flexibility in programming a path for the robot. In the case of the TO GOAL option, the path planning was done completely by the MCS trajectory generator, as only the final goal of joint vectors was passed through the data files. Upon receiving the desired goal position, the CLIF automatically employed the CIRSSE trajectory generator to command the manipulators to the correct joint values. With the VIA PATH option, the user was prompted for an *update rate* which translates to the frequency with which the graphics screen is refreshed. Screen refresh was the key CimStation command that was employed throughout this interface research, and will be discussed more thoroughly in the MOTION menu section. For example, an update rate of 0.25 signals the screen to be refreshed four times a CimStation second, ie. show the progress of the manipulators motion on its way to the goal position, four times for every simulated second. The smaller the value of the update rate, the more the physical arm (ultimately being driven by the MCS software and the trajectory generator by default) will mirror the exact path taken by the CimStation arms.

PREVIEW/CREATE This menu option has no interface function associated with it, and its only purpose is to lead the user to the main CimStation menu where all the move capabilities are located. The user menu only added to the functionality of the CimStation software, providing the necessary interface operations. Any moves of the workcell objects were done from the existing, main CimStation menus. This hierarchy involved substantial travel between the various menus, but was necessary in order to take full advantage of CimStation's move options which could not be accessed directly from the user defined menu.

VIEW STORED Since this interface is considered such an effective path previewing tool, and was created in part for that purpose, the VIEW/STORED menu option is listed. This selection gives the user the opportunity to select the number of robots for which data (prestored joint angle values) will be supplied, and the paths to the data files in which this information has been stored. The data files which are listed, must necessarily be in the appropriate format which the CimStation command [25]

moveto_tabjv(<robot>,<filename>)

is able to decipher; line 1 is an integer which equals the total number of joint angle vectors contained in the data file, line 2 is a real number which denotes the desired update rate and lines 3 -> (3 + line 1) contain the actual joint values in the form of vectors (size n, the number of joints of the robot). Once the choices for robots and corresponding data files have been entered, the concurrent programming function discussed in section 2.1.4, is utilized to move the graphics arms according to the data generated by any external program such as the collision detection algorithm or the trajectory generator. In addition to the advantage of previewing these externally generated paths, the ON mode of the interface can be simultaneously selected, and those paths can command the physical arms indirectly, through CimStation.

<u>ON/OFF</u> Before the ON selection was made, both a workcell type and a robot must have been selected, otherwise the joint information being sent would be meaningless (the supporting code was programmed to safeguard against such an occurrance). The selection of the ON option, prompted the refresh actions function, as specified in the menu code, to be carried out. The refresh actions function, allows for an argument which is itself a function or a procedure. If refresh actions (Figure 3.5) is called with no argument, then each time the screen is updated (depending on the update rate as described above), only the action of redrawing the workcell in its "refreshed" appearance is executed. If however, a previously defined function is passed to the refresh actions, then that function is *called* each time the screen is updated. In the case of this interface, the refresh actions option proved to be invaluable, as it represents an immediate link to the behavior of the graphics robots, and could be harnessed, interpreted and translated to MCS meaningful commands.

STANDARD	**USER DEFINED**
refresh_actions (____________);	refresh_actions (<u>FUNCTION</u>); FUNCTION: - check for robot selection - write the current joint angles of the robot selected to the data file

Figure 3.5 - The Refresh Actions Function

Once all desired motions had been communicated to the testbed, the END SESSION menu selection was entered by the user, incicating that the link between CimStation and

MCS/CTOS should be terminated. The implementation of this was in the form of the string "STOP" being sent to and saved by the first line of the command data file. Once the string was received by the message handler, the process of correctly closing all open data files and disengaging the testbed high power (which was engaged during the FIREUP command) was initiated. With all power down, and data files closed in th UNIX environment, the areas were ready to begin the interface again from the start.

3.4 CTOS Message Handler

As any CTOS task, the interface message handler contained the three bootstrap phases of AINIT, PINIT and AEXEC [19]. The first two phases were not utilized at all, and the body of the code was placed in the AEXEC phase, programmed to switch on three possible cases. These cases were determined by the first line of the UNIX data file which contained all the CimStation string commands. The AEXEC phase was one loop which polled the command file every 1.0 seconds reading line 1 and reporting the contents to the switch.

<u>CASE new</u> If the polling routine of AEXEC returned from the UNIX command data file with the string "new" in its character buffer variable, then the CASE new section of the switch statement was executed. The next three lines of the command file unquestionably had the definitions of the robot type and arm type selected by the CimStation user within them, because as seen above, the four lines beginning with "new"

MessageHandler.c

```
AINIT
PINIT
AEXEC:

 CASE new:
    robot_type = PUMA, PLAT or FULL
    arm_type = LEFT_ARM, RIGHT_ARM
    SEND ---> old

 CASE MOVE:
    PLAT - 3 dof
    PUMA - 6 dof
    FULL - 9 dof    ...... until FINISHED

 CASE STOP:
    TERMINATE INTERFACE
```

Figure 3.6 - The Interface CTOS Message Handler

were directed to the data file in sequence, from the same location in the user menu code. The message handler converted the line 2 and 3 strings into the appropriate variables and immediately cleared the data file (by closing and reopening it) and sent the string "old" to line 1. This method of polling for the strings "new" and "old", was a simple way to ensure that no duplication of effort was undertaken, and that there could be no mistake of the correct robot and arm type being defined.

CASE MOVE The CLient InterFace (CLIF) was the method by which joint angle values obtained from the CimStation refresh_actions sequence were relayed to the Motion Control System. The series of functions which transmit the information were located in the CASE MOVE section of the CTOS message handler, and require only three pieces of

information: - robot type
- arm type
- vector of joint values

which are passed as arguments to the following CLIF function calls:

- clifModeSet - uses default values to establish speed and blending at which the motion will be performed
- clifKnotptSet - defines the goal position (knot point) for that motion
- clifRobotMove - executes the motion using the desired mode and knot point

Both robot type and arm type wer defined in the CASE new section, therefore, this section was dedicated to processing the joint angle information into a readable format which was used in calling the CLIF routines.

The robot type (PUMA, PLAT, FULL) was assigned to a global variable inside the message handler, so that each CASE section would have access to its current value. With the robot type known, the number of joint values expected was also known to be either three, six or nine. The prompt to the message handler given by the MOVE string read from the command data file, was to open the joint data file (Figure 3.3) and read its contents line by line until the string FINISHED was detected. Each line of the joint data file therefore, represented a vector of joint angle values corresponding to the current robot type's number of degrees of freedom (joints). Within the CASE MOVE section, the joint vectors were converted from string variables, as they were read from the data file, to real, radian quantities and assigned to a message handler real array of length three, six or nine. This vector was then in the correct form to be accepted by the CLIF routine clifKnotptSet.
So, with every refresh of the CimStation graphics screen, corresponding to the update rate, a joint vector representing the current value of the robots joints was sent to the joint data file, read by the CASE MOVE section of the CTOS message handler task and processed into correct form for the CLIF to utilize and forward to the MCS resulting in a mirror image motion of the graphical CimStation move.

3.5 Initial Run Results

This first phase of the CimStation MCS/CTOS interface was completed successfully, satisfying the proof of concept trial. Although the mechanics of the scheme were sound, the interface was painfully slow between moves and therefore fated for revision. The reason for the sluggishness of the communication, lay in the inevitable timing incompatabilities between CTOS and the rate at which the data files could be opened, read, and closed again. It is an unfortunate characteristic of the UNIX operating system which does not signal the environment that a file has been updated (changed) unless a command is issued to query that change (ie. a directory listing) or after a default timeout. This update rate is substantially slower than the CTOS message handler was able to poll the joint data file for new joint values, and therefore was forced to wait (poll more times) for the file to signal its refurbishment.

Although it was now clear that an interface between the two processes was possible, a more efficient and timely method was required. The features which were established for enhancement are as follows:

- a more reflective speed of communication
- a more flexible arena for engaging and moving the robots
- an interface guaged more for the less experienced robotics user
- more safeguarding mechanisms to protect hardware and undesirable software crashes

These objectives were evaluated, and their solution was materialized through the implementation of *sockets*.

3.6 Summary

In th method of using data files as a communication link, the two processes, CimStation graphics package, and CTOS message handler code, were interfaced. To support this interface, the physical testbed was modeled using CimStation graphics capabilities, and a CTOS message handler writte to interpret the commands being issued by CimStation. The two had access to common data files, which stored communication information. The interface was successful, but proved to be unacceptably slow.

CHAPTER 4

INTERFACE BY SOCKETS

This phase of the interface was a considerably easier step to take, as the foundation for message passing and the workcell environment had already been defined in the previous phase of this research. As discussed in section 3.5, several areas were identified for improvement after the completion of message passing with data files:

- faster response time of the testbed to CimStation commands
- a method by which robots could be engaged in an arbitrary sequence
- an more user friendly interface
- additional protection against accidental commands

Each of these concerns was assessed, and the ideal solution was determined to be the UNIX function of sockets as a means of passing data. This chapter will describe the message passing sequence used in this phase of the work, and identify and explain the improvements incorporated from the first scheme of message passing by data files.

4.1 Description of Sockets

Sockets are the BSD method for allowing one process to communicate with another in the UNIX environment. Frost [6] describes this inter process communication (IPC) as analogous to a telephone system. Continuing with that analogy (Figure 4.1), a socket connection requires a process designated as the *server* to establish the socket (the telephone line must be installed). The server then waits and accepts connections from other processes (waits for the phone to ring). Another process called the *client*, contacts the server but must know its machine and port number (the caller must have the phone number). For information to be passed from client to server (socket connections are one-way communication), the client executes the routine which uses the portnumber and machine hostname to to contact the server and send the information in a format known to both (the caller dials the phone number, and speaks in the language both understand).

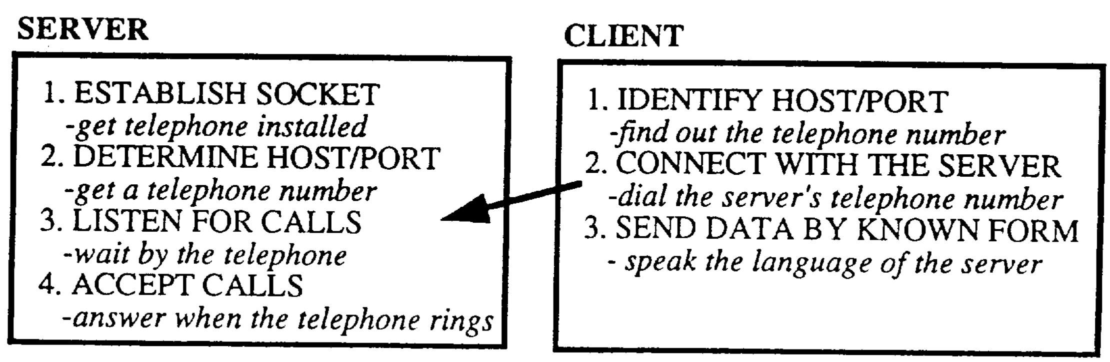


Figure 4.1 - Socket Communication Structure/Telephone Analogy

In this interface research, the server is the CTOS message handler, and the client is the CimStation user menu code. Each performs the same functions as described above by utilizing a library of socket routines, written by Keith Nicewarner, ECSE Department, Rensselaer Polytechnic Institute.

4.2 Logistics of Message Passing

As mentioned above, the framework for message passing was established in the data file phase of research, and in this phase, those commands are simply replaced with socket function calls. Again the CimStation "user defined menu" is utilized to initiate commands which are reflected on the graphics screen, then encoded and sent through the socket structure to the CTOS Message Handler which reads the data, deciphers it and sends the appropriate command via the CLIF functions to the MCS and testbed.

The difference in this method of message passing, is that the Message Handler is no longer polling, because the link to the CimStation user menu is *direct*. This saves a considerable amount of time, in that each process is only engaged when it is either sending or receiving data, and is otherwise idle. This ensures that both processes are in a state of readiness at all times, ie. no time is spent waiting for a process to finish an earlier function, such as polling. The results of this socket method is an almost instantaneous response of the testbed arms to the CimStation commands. Figure 4.2 shows an overview of the socket message passing scheme.

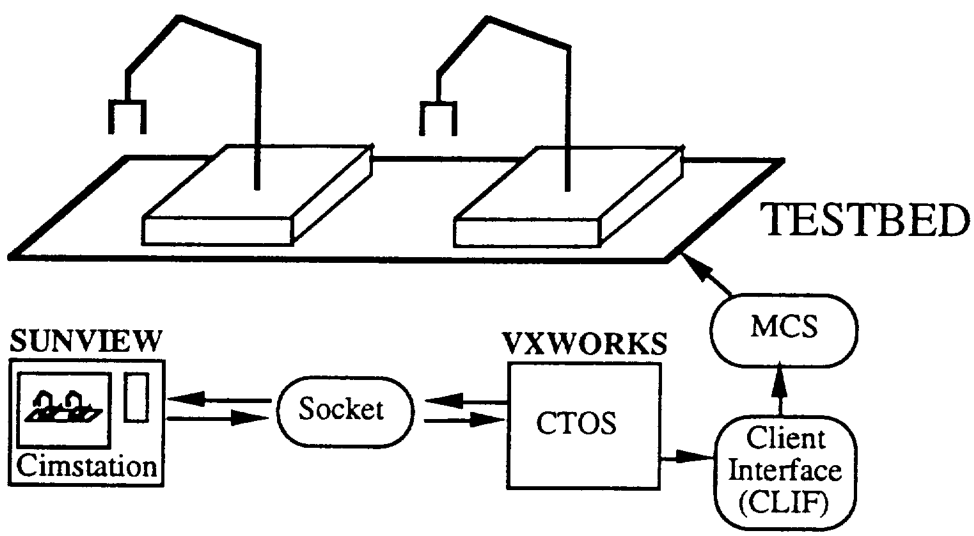


Figure 4.2 - Socket Message Passing Scheme

It appears from this figure that there exists two-way communication between the processes by means of the socket connection, and that the statement made earlier is false, but inter-communication is in fact possible in certain highly controlled circumstances as is shown in section 4.3.1 with respect to robot calibration.

The CimStation code, activated through user menu selections, sends one "message" at a time to the CTOS message handler. Each message is a variable of type *string*, and is one of two message types; a command message for gripper, compliance and quit operations, or a move message for a specific robot. Table 4.1 gives the possible combinations of the command message, and Table 4.2 lists those of the move message.

CHARACTER 1	space	CHARACTER 3	space	CARACTER 5	CHARACTER 6
C - compliance		**O**-open / **C**-close		**P**-PUMA **F**-full arm	**L**-left **R**-right
G - gripper		**O**-on / **F**-off			

Table 4.1 - Command Message Combinations

An example command message would be - "**G_C_FR**", which translates into "close the gripper of the right full arm" which is the same as the gripper of the right PUMA. Note that CimStation will not send command messages for the platform alone, as it has neither a gripper nor can it be engaged in compliant mode. Also, compliant mode is only meaningful in the case of the PUMA arms, so for a command message "x x Fx" or "x x Px" the CLIF will know which *PUMA* arm to make compliant. The quit command message consists of only one character "q", and is encountered only once per session, as this is the command which signals the socket connection to be terminated.

CHARACTER 1	CHARACTER 2	space	CARACTER 5 -->
C-cart P-PUMA F-full arm	L-left R-right		word 1 ... word n n = 3,6,9

Table 4.2 - Move Message Combinations

An example move command is - "**CL_000000..010_000000..011_000000..111**", which translates into, "move the left platform d1=.002°, Θ2= .003°, Θ3=.007°".

The characters of 5+ represent binary words of 32 bits each, in sets of 3,6 or 9 joint vectors. The reason for this representation is that the definition in the socket library routines expects a string "buffer" (variable) to be passed through the socket structure as data. The process by which C code is adapted into CimStation readable code involves several explicit steps, which culminate in the creation of a new CimStation environment (called a template). This process is tediously long, in that template creation takes on the order of one half hour to complete. In addition to this, no clear method exists by which to verify the correctness of the product of these extensive steps, other than the success or failure of the template creation (see also section 1.4).. This awkward method is a process whose execution is purposely minimized, and therefore, instead of a change being made in the parameter lists of the socket routines to accept non-string data (which would require endless trial and error loops to ensure valid results), the original routines were left intact, and the parameter lists (robot specifications, commands and joint angles in this case) were converted to character strings.

SIL has pre-defined all the functions to manipulate real data, such as joint angles, into *word* (32 bit binary representation) format, and consequently, the C language can be programmed to decode that format. Because the word representation is compatible to both languages, it was selected as the means by which joint angle information would be communicated through the socket structure. Figure 4.3 shows the procedures which are followed to ensure successful communication: CimStation obtains the joint angles through the refresh actions sequence (as before) and then multiplies the value by 1000 and truncates to obtain an integer value. The coefficient of 1000 is chosen because the physical robot encoders are only accurate to 0.005 degrees, and therefore any precision beyond that would be irrelevant. On the client (CTOS) side of the socket, the binary digit is processed as shown to retrieve the original joint angle, and then converted to radians, which is the format the CLIF routines expect.

SIL Function	**Example**
Θ_i [deg]	30.5°
Θ_i * 1000	30500.000
real -> integer truncation	30500
integer -> ***binary word***	00000000000000000111011100100100

SOCKET

C Function	**Example**
binary word -> real	30500.00
Θ_i + 1000	30.50000
degree to radian conversion	0.53232
Θ_i [rad]	0.53232 rad

Figure 4.3 - Encoding and Decoding of Joint Angle Information

4.3 CimStation User Menus

The structure of the socket interface menu is not very different from that of the data file interface menu. Most menus and submenus were left unchanged, some were enhanced to incorporate the objectives of section 3.5, and others were added to incorporate the new socket structure and new features. Figure 4.4 shows a hierarchy of the user menus, and the remainder of this section will be devoted to the explanation of each of those menu and submenu choices.

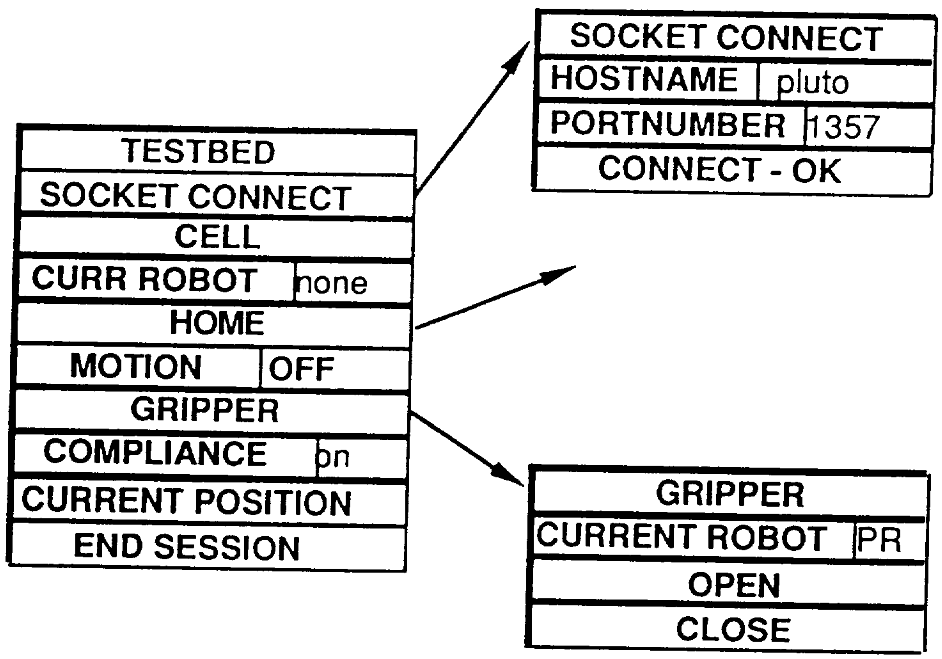
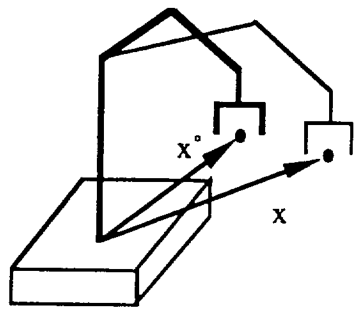


Figure 4.4 - User Menu Hierarchy

Several menus have not changed from the previous phase, for example CELL still prompts the user to select wether she will be engaging two 9-DOF arms, or two 6-DOF PUMAs plus two 3-DOF platforms. The fact that CimStation can only command one robot at a time while not in concurrent programming mode (section 2.1.4), has not changed.

The MOTION menu has undergone only two enhancements. The first of which involves the refresh actions function (MOTION/ON) only sending joint angle data through the socket, if the change in the joint vector from the previous position is:



$$\| x^{\circ} - x \| \leq (1^{\circ} * n) = (.01745 \text{ rad} * n)$$

where: $x^{\circ} \equiv$ original vector of joint angles
$x \equiv$ vector of joint angles at goal position
$n \equiv$ number of degreees of freedom

and each joint has made a change of at least 1° (0.01745 rad). This function calculates the norm of the two vectors (x° and x), and tests wether it is less than the number of DOF times one degree. That is, unless each joint has moved more than 1 degree, the new position is not sent. The linear joint has units of centimeters in CimStation, therefore the value of "1" in the first joint corresponds to 1 cm, but the same weighting is given to it as the rotational joints. Since the CimStation image is refreshed at every menu choice, traveling between menu trees will cause duplicate joint angles to be sent through the socket unnecessarily. This vector norm calculation suppresses that extraneous data, but, if a very small move is pruposely desired, then the goal will not be sent. To remedy this, the testvalue in the norm equation should be decreased. The second change in the MOTION menu is also in the refresh actions function (Figure 4.5), whereby the type of robot currently engaged, is concatenated in front of the joint angle data. The form which this designation takes, is two capital letters (see Tables 4.1 and 4.2 for details), the first describing the robot type (cart, PUMA or full arm), and the second describing the arm type (left or right).

STANDARD	USER DEFINED
refresh_actions (_______);	refresh_actions (FUNCTION); FUNCTION: - determine which robot is engaged (CL,..., FR). - check if enough motion has been seen by each joint (norm calculation) - concatenate the current robot string (CL,..., FR) with the word representation of the joint angles.

Figure 4.5 - The Updated Refresh Actions Function

As defined in section 3.3.2, the refresh actions function is executed every time the graphics screen redraws itself, which is every time a menu selection is made or the robot joint angles have been changed.

4.3.1 Main Menu Items

The main user menu contains several new additions which do not call on individual submenus, and each will be described here.

Current Robot This function was developed as an internal check variable, which is *always* set to the name of the robot that the user desires to manipulate. The advantage of this routine is that the internal safeguards against accidental movement of a robot not consistent with the testbed's situation, are greatly simplified. By knowing the type of robot engaged, the menu code is able to check for errors at every subsequent menu choice, for example; if CURR ROBOT is CR, and gripper open is selected, then the code knows to send the message "No gripper compatible with message" to the screen. This is a feature implemented to satisfy the criterion of "additional protection against accidental commands" set out in section 3.5 as a desirable option. The disadvantage to this scheme, is that the user must faithfully remember to change (or at least check) the "current robot" menu button, which keeps the current robot information displayed continuously, before every session of sending commands through to CTOS. Unfortunately there is no function in CimStation which would limit a user's access to a particular robot within the cell, so the stipulation is made to the user, that she pay close attention to the current robot display feature. The same convention for defining the robot and arm type is used throughout this interface as follows:

ROBOT TYPE	ARM TYPE
C - platform, 3-DOF	L - left
P - PUMA, 6-DOF	R - right
F - full 18-DOF	

There are six possible combination of the above characters, which represent the six types of robots which can be engaged for motion: CL, CR, PL, PR, FL and FR.

Compliance The compliance mode is a function of the MCS system which permits the PUMA arms (the only DOF for which compliance is defined) a certain controlled flexibility when they come into contact with a rigid surface or external force. This application is used when the arms are operating in an environment which has multiple obstacles or is in a teleoperational mode, to allow for uncertainties which plague the path planning of any manipulator. The menu choice "compliance" simply checks the compliance setting and either obeys the user's command (on/off), or gives an error message that there is no current robot, or that compliance is already on/off.

Current Position This menu option was developed primarily as a hook for future work, but has a function in this interface also. The goal here is to determine the state of the physical testbed (ie. value of all the joint angles as the hardware/sensors perceive it), and send that information to CimStation, which translates it into a "start" position of the testbed. This is the issue of world model vs. physical workcell calibration which will be discussed in Chapter 5.

The socket communication link has been defined as "one-way" but there is one exception to that definition; if the client process (which is normally *sending* the information to the socket) issues a command which is interpreted by the server as a request for a response, and the next line following the command issuance in the client process is a function which will accept the server's response, then a read function may be performed, capturing the response information. This data, once internal to the client code, can easily be processed as before. This seems like a long list of criteria to implement two-way communication in a scheme defined as uni-directional, however it is an intuitive process. Building on the telephone analogy of section 4.1, the client (caller) asks the server (listener) a question, and communication between them does not resume until the listener has provided the caller with a response. Sockets are equipped with message queuing capabilities, therefore, it is reasonable to require communication between client and server to be suspended while a response is being formulated.

The application of this functionality in the interface, is that only one socket direction need be established, and when the CimStation process requires current position information of the message handler, a format as described above is implemented.

4.3.2 Submenu Items

There are three submenus in the new version of the interface which will be discussed in this section.

<u>Socket Connect</u> As described in section 4.1, the CimStation software (specifically the user menus) is the *client* to the socket connection, which provides data (in the form of a string variable here) to the server, represented by the CTOS message handler. The server establishes the socket through a library of routines, by prompting the user for an arbitrary

integer greater than 1000. This integer then, in addition to the chassis name are manually typed at the PORTNUMBER and HOSTNAME submenu options. When these have been input, and are correct, the CONNECT - OK button is chosen, and the link routines execute to complete the link. Now a functioning socket exists, and is identified uniquely by the hostname and portnumber. The supporting C code which was incorporated into CimStation is described in section 4.3.4 (source code is in Appendix C).

Gripper This submenu simply operates the testbed grippers, open and close, and sends the same command through to the CLIF. CimStation provides the user with a separate menu to operate the end effectors of the workcell, but these are commands internal to CimStation and would not actuate the physical grippers. Therefore, motion of the grippers is desired in the physical system, then their operation must be commanded from the TESTBED/GRIPPER menu.

Home Again, this submenu is a duplication of the home command which can be issued out of the CIMSTATION/LAYOUT WORKCELL/ROBOT PENDANT menu, but the functions defined in the TESTBED/HOME menu are specific to the CIRSSE testbed, in that all robots within the cell can be sent back to their home or zero positions (defined in Table 3.1). This is simply a convenience menu, and can be utilized to actuate the graphics screen only, or in combination with the TESTBED/MOTION/ON option will echo those moves to the physical testbed.

4.3.3 C Code in CimStation

As described in section 4.2, the integration of C code into the CimStation/SIL [27] environment is not a trivial task. The requirement on the C code, is that it be in a format which the SUN compiler can process. This not ANSII C format, and therefore some alterations to the socket library functions were made, (eg. there is not function prototyping for the SUN C compiler). With this code saved in the appropriate directory, the corresponding SIL code is developed as a wrapper around the C code, that is, the SIL program (eg. user menu code) calls the SIL function do_this(); do_this in turn is a function which imports the SIL compiled C code, and calls the C code. During execution, the SIL wrapper, function do_this() is called, the imported C code is triggered and returns a value, which is then passed back through the SIL function to the calling SIL routine.

In order for the SIL wrapper to know where to look, and so that the external C code does not have to be recompiled with each function call, this compilation is done ahead of time, linking the appropriate routines. These compiled results are stored in a library , and incorporated as an INPUT/OUTPUT option for the session environment (template), which is created once, and then simply installed at start of a CimStation session. For more information about integrating C code into CimStation see [27] and also cprogram.notes written by Steve Murphy as a supplement to the Silma literature.

4.4 CTOS Message Handler

The CTOS Message Handler is the C code referred to as a task in CTOS parlance, which manages the data and messages with which it interacts. Figure 4.6 shows the message handler for this interface by sockets phase of research. The organization of this

MessageHandler.c

```
AINIT
PINIT
AEXEC:
  prompt user for portnumber -> establish socket for client to conect with
  while the message is not 'q':

  CASE Cx words(3):
      move the left/right CART - 3-DOF

  CASE Px words(6):
      move the left/right PUMA - 6-DOF

  CASE Fx words(9):
      move the left/right FULL arm - 9-DOF

  CASE G x xx:
      open/close the left/right GRIPPER

  CASE C x xx:
      turn COMPLIANCE on/off in the left/right puma

  CASE q:
      terminate socket connection
```

Figure 4.6 - The Updated CTOS Message Handler

code is much more modular than its predecessor, and consequently more straightforward. As shown in Tables 4.1 and 4.2, there are six possible types of messages which can come through the socket, three robot move commands, a gripper command, a compliance command and a command to quit. Those are the functions that this research has been limited to, but with the modularity of the code, others could be added with little extra work.

At the first call of the message handler, the full 18-DOF are engaged for motion, and a "key" (CLIF integer representation for robot reservation) is obtained. Then, a clifRobotSplit() is called to split the original key into two separate ones, each overseeing one half (9-DOF) of the testbed, denoted as left and right keys. With these established, the user is free to actuate any robot in the graphical testbed in any sequence (provided the "current robot" selection is made first, to ensure proper error checking within CimStation). This is a considerable improvement over the scheme used with data files, because there, each robot was necessarily engaged separately before each motion, which impeded the progression of the algorithm. This is not a sound course of action from the point of view of the CLIF. The method of engaging the full testbed and then performing a key split to obtain "rights" to each 9-DOF separately is ta better method of operation.

With the socket established, the testbed engaged, and keys defined, the interface is ready to proceed. At this stage, the user travels within the CimStation menus and the User menus, utilizing the package to her own specifications. The messages generated by CimStation are transmitted through the socket and interpreted by the message handler which reads the first character of the string, and passes the message to the appropriate function, switching on the first character. The motion commands (first character C, P, or F) involve triggering the correct function call, defining the current position of the testbed (via a joint angle query CLIF function), and defining the joint vector which will be passed on to the move function, which is common to all three motion commands, and accepts a joint vector of size 9. This parameter size allows for the generality of the function, the only stipulation is that the move functions pass the correct array size as the joint vector (9).

Each move function therefore, depending on the first character of the message string, knows how many joint angles to expect and which angles they will be. For example, a command of "Px word1...word6" triggers the PUMA move function to convert the six joint values from binary words, to real numbers to radian values, and insert three platform joint angles (obtained from the CLIF joint angle query function) before them. This procedure is similar for the other two move functions. This flow of logic is performed upon the arrival of every CimStation message.

In the event that the CimStation message is a command, then the CTOS message handler recognizes this by reading the first letter of the string, and comparing it to G, C or

q. In the case of the first message being G x xx, the gripper function is called, and the third character is evaluated to be an "O" to open the gripper or "C" to close the gripper of the robot defined by characters five and 6. The same applies for compliance commands; The string "C x xx" is evaluated by the compliance function, and the third character is evaluated a "O" defining compliance be turned on, or "F" defining compliance to be turned off. Again, the robot under consideration is defined by characters five and six. The commands are screened inside CimStation, that is using the *current robot* feature, the validity of choices is determined before any information is sent through the socket structure.

As mentioned earlier, this algorithm is organized in such a way that will make the subsequent addition of features extremely simple. The format is:

- a user menu is created inside CimStation [27] for the desired command
- the code is equipped with a sufficient amount of error checking provisions so that illogical information is not passed to CTOS as a message
- an identifying character is defined for that operation which the message handler switches on when the message is receive through the socket structure

4.5 Improvements Made and Those Yet to be Made

The four areas which were identified at the conclusion of Chapter 3, were all incorporated into the second phase of this interface research:

• "a more reflective speed of communication" was achieved simply by implementing UNIX socket functions as the medium of interaction. Because the communication between processes is direct, no time was expended waiting for the information to be available.

• "a more flexible arena for engaging and moving the robots" was created by engaging the full 18-DOF of the testbed at the start of the interface session, and delegating key integers to each half (9-DOF) of the testbed. With each CimStation command being encoded with the robot type and arm type of the robot under consideration, the flexibility the user has in choosing and moving the robots is greatly increased.

• "an interface guaged more for the less experienced robotics user" and "more safeguarding mechanisms to protect hardware and undesirable software crashes" were both ensured with added messages, and more/better safeguarding functions against the inevitable errors which an inexperienced user is bound to encounter. The logic behind the interface was revamped, in order to seem more intuitive to the novice user.

Though this interface using sockets is far superior the the first attempt using data file as an intermediary, there are still areas which could be improved. Chapter 5 discusses the very broad direction in which this research should continue, whereas the remainder of this section will concentrate on areas particular to this application which deserve to be improved.

• the 32 bit binary word format used to communicate joint angle values from CimStation to the CTOS message handler were represented as such to conserve the effort involved in altering the socket routines. If it was concluded that transmitting data by real numbers for example was more efficient, then this upgrade would deserve implementation.

• the subsequent version of the CimStation software (version 4.3.1) contains a calibration package in addition to several other features which could be exploited. This version, although available, was not implemented due to insufficient memory capabilities.

• the scheme by which joint values arriving through the socket structure are being concatenated with the "current" position of the testbed is not ideal. By querying the testbed prior to every goal vector move, and substituting those values into the array locations for which CimStation has not provided data (ie. the first three joints are undefined in the case of a PUMA's motion), the interface algorithm overrides the positioning loop which supports the controller code, and subsequently causes the queried joint angles to physically "drift". This situation could be remedied by only utilizing the CLIF joint angle query function once per a robot type's sequence of motion.

• the final test of this interface, of course would be its implementation in a telerobotic setting. The use of a video camera or other informational source could provide verification of the interface's accuracy.

CHAPTER 5

FUTURE WORK

The work which has been accomplished with respect to the CimStation graphics package and CIRSSE testbed, represents a world modeling scheme and an off-line programming tool. These concepts were described briefly in section 1.2, and areas where similar research is being done were cited. In this chapter, calibration is discussed as the outstanding issue which would completely define the world model already developed with Cimstation.

5.1 Calibration Overview

Silma Inc [24] has introduced an additional package which is supported by CimStation version 4.3.1. The main purpose of this and any other calibration scheme, is to equate a graphical model with the physical model being "uncalibrated" (ie. veering from kinematic and dynamic definitions) by physical constants such as gravity and the unavoidable existence of manufacturing uncertainties. The only means by which an artificial system could hope to represent a physical system, is with an adequate supply of information about the system being modeled. After all, if everything is known about a system, then it can be modeled exactly. The type of information which is beneficial (if not crucial) to the success of a calibration algorithm, is that which comes from the various sensors mounted around the workcell.

5.2 Calibration Techniques

Nowrouzi, et al [18], have compiled an overview of the most widely used calibration techniques. Each to some degree, measures the robots static and dynamic performance, and compares it to the factors of idling servoing behavior, accuracy, repeatability, over/undershoot, cross sensitivity and settling time. The evaluation of this comparison yields the calibration techniques of the Cube Method which engages a positioning sensor over time, the Laser Tracking Method which uses laser refraction to determine position, the Three-Cables Method utilizes information about the lengths of three cables attached to the robot's wrist to determine position, and the Ball-Bar Method which attaches a machine tool to the robot's end effector, and mechanically moves the tool in specified orientations, and the manipulators response determines the degree of calibration. The fifth and final technique described the Theodolite and TV Camera Method, utilizes camera vision to determine position information. This is the method of choice for many research and industrial settings (CIRSSE included) as the technology is relatively inexpensive, and results are accurate, especially with multiple cameras. The size of an image taken by the camera is directly related to the proximity of the object, and therefore a series of comparisons yield the distance of the object from the camera. This information is used to determine position and orientation .

The techniques mentioned above represent the general pool of resources which can be implemented for calibration of a physical system with a world model. AREEM another calibration technique, developed by Tunstel and Vira [31] is a program to improve position accuracies by providing scalar algebraic equations which represent the positive error correction between the world and physical models. Renders, Rossignol et al [22] have created a calibration method which identifies differences between real and world models by a maximum likelihood approach to identify geometric errors. No matter what the calibration technique, each is heavily dependant on information obtained form the workcell sensors.

5.3 Types of Sensors

For successful world representation of a manipulator system, there is a need for one or more types of sensory information to be available, which through innovative combinations can coerce the world model to approach a representation of reality. The standard type of sensors which are readily available, resemble closely the human senses:

- tactile sensors
- proximity and range sensors
- temperature and presence sensors
- machine vision sensors [7]

Research continues on more state of the art sensors which are more complex, to maximize the amount of information which can be transmitted by them. An example of a sensory vehicle which is being developed by Jau [11], involves a telepresent human-like hand system. Again a sensory system which is modeled after the human senses and configurations. Additional research is being done by Wang [32] with a vision sensor mounted on the manipulator of a testbed to perform extrinsic calibration. This setup is very similar to the wrist cameras of the CIRSSE testbed and the cameras mounted on the body of the Space Shuttle Arm.

No matter which calibration technique is selected, the necessity for multiple, accurate sensor devices is clear. The ideal scheme to process the sensory data is not well defined, but as shown here, several prototypes for initial calibration methods are available. In the case of this research, the calibration of the testbed with respect to itself and to the world has been explored. For calibrating the physical testbed to the CimStation testbed, the sensors which are already in place (cameras, lasers, force/torque) would have to be exploited. The sensors which are present in the testbed, in combination with enough of the calibration techniques listed here, should be sufficient to synchronize the two worlds.

CHAPTER 6

CONCLUSION

The goal of this thesis research, was to make an interface between a graphical display, and the physical model. The first step in accomplishing this, was to create an accurate model of the world, the CIRSSE testbed. This was done using the CimStation graphics package as a tool, and was chosen because of its extensive robotics capabilities, and functions. Each component of the physical workcell, the three DOF Aronson platforms, the six DOF PUMAs, the full one half of the testbed (nine DOF), the CIRSSE grippers (left and right) and finally the strut rack and strut were modeled with CimStation primitive shapes (ie. cylinders, blocks, etc.). These were then assigned the appropriate kinematic parameters, and identified as robots, grippers or objects. All the modeled elements of the testbed were grouped into the correct configuration, and labeled as the "modeled world".

A graphical interface between the CimStation graphics package and the CIRSSE testbed was then accomplished through the implementation of the MCS/CTOS system of process communication. Two methods were tested to create interaction between the simulated workcell and the physical model, the first was an interface using UNIX data files whose Input/Output capabilities are accessible by both processes and the second is the current interface, which uses BSD sockets as the link for message passing. The latter was found to have a much faster response time than the former, on the order of 1 second versus 30 seconds. The socket interface also proved to be more intuitive to a novice user, it was programmed with several more error detection capabilities and showed good promise for future upgrading. This interface could be a useful tool in the areas of telerobotic manipulation, path previewing and off-line programming.

This interface makes the assumption that the world model and physical model are exactly the same (or that knowledge about the position of one necessarily defines the position of the other), and that during motion this assumption holds true. This is a valid assumption for the development of a working interface, but must be re-evaluated when implementing the interface for physical tasks and paths to incorporate the idea of uncertainty in calibration and measurements. The issue of calibration between a world model and the physical model is identified as the direction for future work in this area.

LITERATURE CITED

[1] Bedworth, D.B., M.R.Henderson and P.M Wolfe, Computer Integrated Design and Manufacturing, Mcgraw-Hill Book Company, New Yorkn, NY, 1991.

[2] CIRSSE, "Center for Intelligent Robotic Systems and Space Exploration, Rensselaer Polytechnic Institute", Informational Literature, Troy, NY, 1991.

[3] Conway, L., R.A. Voltz and M.W Walker, "Teleautonomous Systems: Projecting and Coordinating Intelligent Action at a Distance", IEEE Transactions on Robotics and Automation, vol. 6, no. 2, April 1990.

[4] Craig, John J., Introduction to Robotics, 2nd Edition, Addison-Wesley Publishing Company, Reading, MA, 1989.

[5] Fieldhouse, K.R., "A Computing Guide at CIRSSE", CIRSSE Technical Information, November 1990.

[6] Frost, Jim, "BSD Sockets: A Quick and Dirty Primer", Software Tool & Die, November 1989.

[7] Groover, Mikell P., Mitchell Weiss, Roger N. Nagel and Nicholas G. Odrey, Industrial Robotics - Technology, Programming, and Applications, McGraw-Hill Book Company, New York, NY, 1986.

[8] Han, Ding and Lou Lin Huang, "Computer-Aided Off-Line Planning of Robot Motion", Robotics and Autonomous Systems, vol. 7, no. 1, March 1991, pp. 62 - 72

[9] Hayati, S. and J. Balaram, "Supervisory Telerobotics Testbed for Unstructured Environments", Journal of Robotic Systems, vol. 9, no. 2, March 1992.

[10] Hayati, S., T. Lee, K. Tso, and P. Backes, "A Testbed for a Unified Teleoperated-Autonomous Dual-Arm Robotic System.", Proceedings of the IEEE International Conference on Robotics and Automation, Sacramento, CA, vol. 2, May 13-18 1991. pp. 1090- 1101 .

[11] Jau, Bruno M., "Man-Equivalent Telepresence Through Four Fingered Human-Like Hand System", Proceedings of the IEEE International Conference on Robotics and Automation, Nice, France, vol. 1, May 12-14 1992, pp. 843-848.

[12] Kernighan, B. and D. Ritchie, The C Programming Language, 2nd Edition, Prentice Hall, Englewook Cliffs, NJ, 1988.

[13] Kim, W.S., P.G. Backes, S. Hayati, and J. Bokor, "ORU Changeout Experiments with a Telerobot Testbed System", Proceedings of the IEEE International Conference on Robotics and Automation, Sacramento, CA, vol. 3, April 9-11 1991, pp. 2026- 2031.

[14] Lefebvre, Don, "CTOS Tasks and Applications", CIRSSE Technical Memorandum #5.1, Rensselaer Polytechnic Institute, Troy, NY, August 1992.

[15] Lefebvre, Don, "User Guide to CTOS Message Passing", CIRSSE Technical Memorandum #6.1, Rensselaer Polytechnic Institute, Troy, NY, September 1992.

[16] Lefebvre, D.R., L. Page and J.R. Noseworthy, "CTOS Course", CIRSSE Report #128, Rensselaer Polytechnic Institute, Troy, NY, November 1992.

[17] Mazer, Emanuel, et alii, "ACT: A Robot Programming Environment", Proceedings of the IEEE International Conference on Robotics and Automation, Sacramento, CA, vol. 2, April 9-11 1991, pp. 1427-1432.

[18] Nowrouzi, Y.B., H. Kochekali and R.A. Whitaker, "An Overview of Robot Calibration Techniques", The Industrial Robot, December 1988, pp. 229-232.

[19] Page, Lance, "Introduction to Using CTOS on Unix", CIRSSE Technical Memorandum #16.1, Rensselaer Polytechnic Institute, Troy, NY, April 1992.

[20] Park, Jonk H. and Thomas, B. Sheridan, "Supervisory Teleoperation Control Using Computer Graphics", Proceedings of the IEEE International Conference on Robotics and Automation, Sacramento, CA, vol. 1, April 9-11 1991, pp. 493-498.

[21] Ravani, Bahram, "World Modeling for CAD-Based Robot Programming and Simulation", International Journal of Robotics and Automation, vol. 4, no. 2, 1989, pp. 96-105.

[22] Renders, Jean-Michel, et alii, "Kinematic Calibration and Geometrical Parameter Identification for Robots", IEEE Transactions on Robotics and Automation, vol. 7, no. 6, December 1991, pp. 721- 732.

[23] Sheridan, Thomas B., "Teleoperators, Telepresence and Telerobots", Undersea Teleoperators and Intelligent Autonomous Vehicles, January 1987, pp. 3- 15.

[24] Silma Inc., "CimStation Calibration Package, Version 1.1", Release Notes, Version 4.3.1, Silma Inc., Cupertino, CA, January 1992.

[25] Silma Inc., "CimStation Command Reference", Version 4.2.1, Silma Inc, Cupertino, CA, 1991.

[26] Silma Inc., "CimStation SIL Language Manual", Version 4.2.1, Silma Inc., Cupertino, CA, 1991.

[27] Silma Inc, "CimStation User's Guide", Version 4.2.1, Silma Inc., Cupertino, CA, 1991.

[28] Smith, Michael, "An Environment for More Easily Programming a Robot", Proceedings of the IEEE International Conference on Robotics and Automation, Nice, France, vol. 1, May 12-14 1992, pp. 10-16.

[29] Swift, Dave, "Kinematic and Dynamic Parameters for the Testbed Grippers and Loads", CIRSSE Technical Memorandum #14.1, Rensselaer Polytechnic Institute, Troy, NY, January 1992.

[30] Tendick, Frank, et alii, "A Supervisory Telerobotic Control System, Using Model-Based Vision Feedback", Proceedings of the IEEE International Conference on Robotics and Automation, Sacramento, CA, vol. 3, April 9-11 1991, pp. 2280-2285.

[31] Tunstel, Ed and Noren Vira, "Computer Generation of Geometrical Error Equations Applicable for Improvement of Robots' Positioning Accuracy", Robotics and Autonomous Systems", vol. 7, no. 1, March 1991, pp. 3-26.

[32] Wang, Ching-Cheng, " Extrinsic Calibration of a Vision Sensor Mounted on a Robot", IEEE Transactions on Robotics and Automation, vol. 8, no. 2, April 1992, pp. 161-175 .

[33] Watson, Jim and Steve Murphy, "Testbed Kinematic Frames and Routines", CIRSSE Technical Memorandum #1.3, Rensselaer Polytechnic Institute, Troy NY, October 1992.

APPENDIX A

USER'S GUIDE

CIMSTATION - TESTBED INTERFACE

This guide is intended for those users who have some knowledge about CTOS and the CIRSSE computing system. References will be made to names of workstations and CTOS commands with the assumption that the reader is familiar with the terminology, or has the resources to learn about it. Also, the assumption is made that the user of this interface has the necessary access to the testbed facilities, computing facilities and is set up to run the CimStation software.

Requirements

The requirements for implementing the testbed interface, in addition to those specified above, are:

- the use of two workstations, the CIRSSE testbed and the VMEbus cage is necessary.
- One of the above mentioned workstations must be a Sun 4, with the CimStation software installed.on it, and the other can be any workstation in the testbed lab.

To Start CimStation

- Log into SunView, and at a unix shell prompt, type "cimstation" or the command to start the software if this one should change.

- The CimStation menu appears, and the user should select option 1 - Start CimStation:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* *                                                           * *
* *                  CimStation Version 4.2                   * *
* *                       SILMA, Inc.                         * *
* *                     --------------                        * *
* *                     Copyright 1991                        * *
* *                   All rights reserved                     * *
* *                                                           * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


 ___________________________________________________________________
|                                                                   |
|                 *** CIMSTATION MAIN MENU ***                      |
|                                                                   |
|    (1)  Start CimStation           (2)  Manage Templates          |
|    (3)  Manage User Areas          (4)  Manage Project Areas      |
|                                                                   |
|___________________________________________________________________|

Option or [return] -> 1

Active templates:

cirsse_sock1
cirsse_sock2
weaver

Enter template name or [return] -> cirsse_sock1
```

- The active templates will be listed, and the one which should be typed at the enter template prompt is cirsse_sock1, which stands for sockets 1.
- CimStation will now load the software, which usually takes about 5 minutes.
- The CimStation window is a full screen window, and the default object is the "teacher", which is simply a reference frame with tetrahedral lines at the origin to identify orientation, and is used for modeling purposes.
- Next, the interface user menu must be loaded into CimStation's memory with the command: sil_load('/home/hron/cim/sil/menu.sil');

A series of T, TRUE and ok messages will be displayed in the command window at the lower left of the screen. These are the compiler messages signifying the successful compilation of each internal structure, and are a good check that the code is being loaded correctly.

• When command window shows ok
ok , this signifies that the user menus are ready to be utilized.

• In the main CimStation window (top right), select the APPLICATIONS button, this will bring up the menu for which code was just installed titled TESTBED.

To Start CTOS

• Log into xWindows, and cd to /home/hron/ctos/testbed/grinc/ or the directory where the files:

files:	Imakefile	sgrincLib.c
	sgrinc.cfg	sgrincMsgHandler.c
	sgrinc.h	

are located.

• The workstation at which the CTOS task (sgrincMsgHandler.c) will be running, should be reflected within the sgrinc.cfg (configuration) file. Three lines inside the .cfg file the chassis name of the workstation:

```
chassis vx0 1  /usr2/testbed/exp/vxworks/demos/clif/clif.cfg_fast
chassis mercury

sequencer mercury

PREFIX mercury 0

%===== LOAD CODE =====

%0 echo LOADING GRINC...
chdir /home/hron/ctos/testbed/grinc
task sgrinctask sgrincMsgHandler 111
```

and should be change from mercury to the correct name. If the file is changed, then inside that directory, a cmkmf all command should be given to recompile the code.

• This interface uses the experimental tree, and therefore, this directory tree must be so indicated on the VT terminals at the CTOS screen (Experimental). If this is not the case, from an xWindow prompt, type ctoskernel, and follow the menu commands to change the directory tree to experimental. This action will re-boot the cage, which prepares it to be engaged. This should be done manually if the CTOS screens are not displayed.

• Now, the CTOS application is started: from inside the same directory, type

-> app_win *chassis name* ↵

-> app_bts sgrinc.cfg ↵

The first command initiates an application window which will scroll hundreds of messages which can be useful in the case of a malfunction, but may be neglected otherwise.

• The "RecWindow" will appear, and during the AEXEC stage, the testbed is engaged, so the controllers, platform, grippers and VME cage must all be on. During this phase, there will be a prompt to **Turn on High Power** of the left and right arms.

• Next, the RecWindow will prompt the user for a portnumber. This will be the identifying number for the socket connection, and therefore must be remembered, so that the ***same*** number is entered in CimStation to complete socket connection. This number should be greater than 1000, generally between 1200 and 1800 is a good choice. For example ... ENTER portnum: 1357 ↵

• This concludes the work at the xWindow terminal, and the rest of the interface is conducted completely from CimStation.

Interfacing

• Selecting the APPLICATIONS button in the CimStation main menu, will bring up the interface menu TESTBED

TESTBED	
SOCKET CONNECT	
CELL	
CURR ROBOT	none
HOME	
MOTION	OFF
GRIPPER	
COMPLIANCE	on
CURRENT POSITION	
END SESSION	

• First select SOCKET CONNECT and enter the information about where the CTOS task is running, and the portnumber which was entered in the RecWindow.

• Select CONNECT-OK, which will complete the socket connection, and allow the in interface to continue.

• The next menu choice should be CELL, which prompts the user for one of two cells:

2x9 DOF
2x6 DOF + 2x3 DOF

Please read section 3.3.1 to determine which cell should be selected.

• From this point forward, it is the user's choice as to which menu functions to engage. The basic flow of the interface, is that a motion is defined using the CimStation menus, like CIMSTATION/LAYOUT WORKCELL ... and potentially stored using the programming capabilities. Then, when the path has been refined to the user's preferences, transmission to the testbed arms may begin.

To Transfer Motion Information

• Define the path either with the programming menus in CIMSTATION/PROGRAMMING or interactively once the interface has been made.

- from TESTBED/MOTION, follow the messages, and when ready select ON.
- from this point forward, <u>any</u> move made on the graphics screen by the robot will be reflected in the physical testbed!!
- the types of motion which can be transmitted are a pre-saved path, which is simply <run> after the ON selection, interactive motion which involves the user employing the CIMSTATION/LAYOUT WORKCELL/MOVE or ROBOT PENDANT selections or the use of external data (from collision detection or elsewhere). This option is chosen out of the TESTBED/MOTION/ menu, under VIEW STORED.option.

MOTION
TO GOAL
VIA PATH
PREVU/CREATE
VIEW STORED
ON
OFF

General Instructions

• **ALWAYS** choose the TESTBED/ CURRENT ROBOT button before doing any manipulation with the testbed and especially before transmitting data. The interface relies on the knowledge of this information for most of its functions.

• Once a current robot is selected, this will be shown in the menu for reference. When manipulating the arms through MOVE commands in the CimStation trees, the option will be given to move any of the robots in the cell. Be sure to only choose the robot which has been previously selected as the current robot!

APPENDIX B

CIMSTATION WIREFRAME MODELS

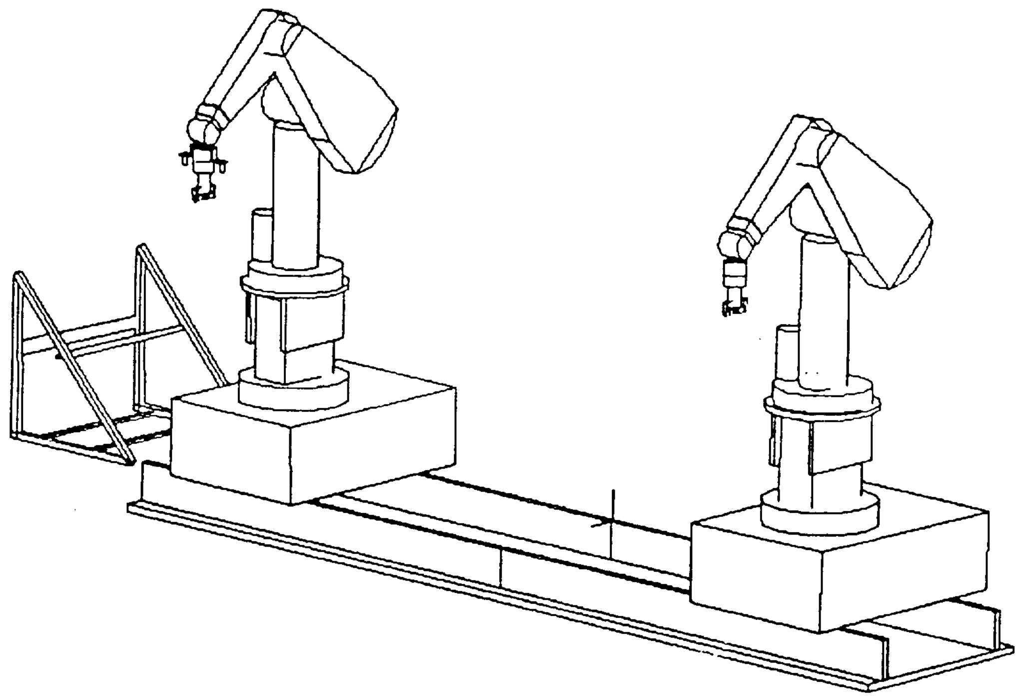


Figure B1 - CIRSSE Testbed

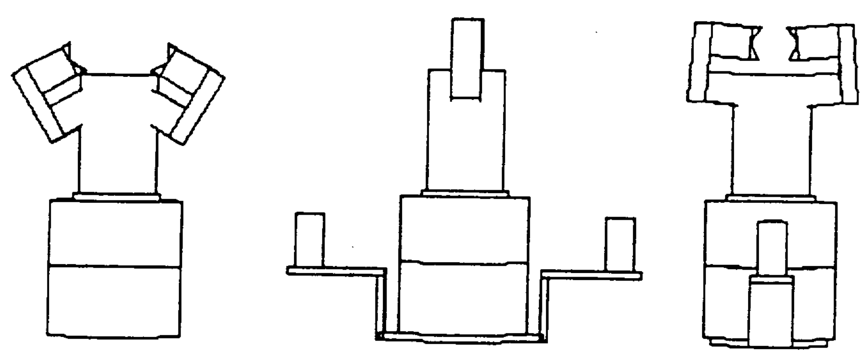


Figure B2 - CIRSSE Grippers-Right (A) and Left (O)

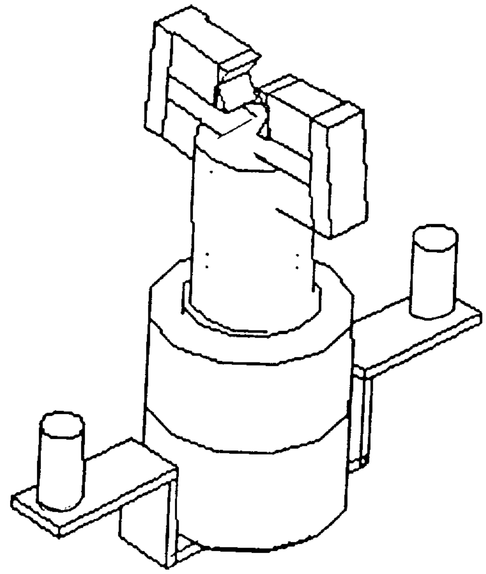


Figure B3 - Left Gripper (O) With Camera Mounts

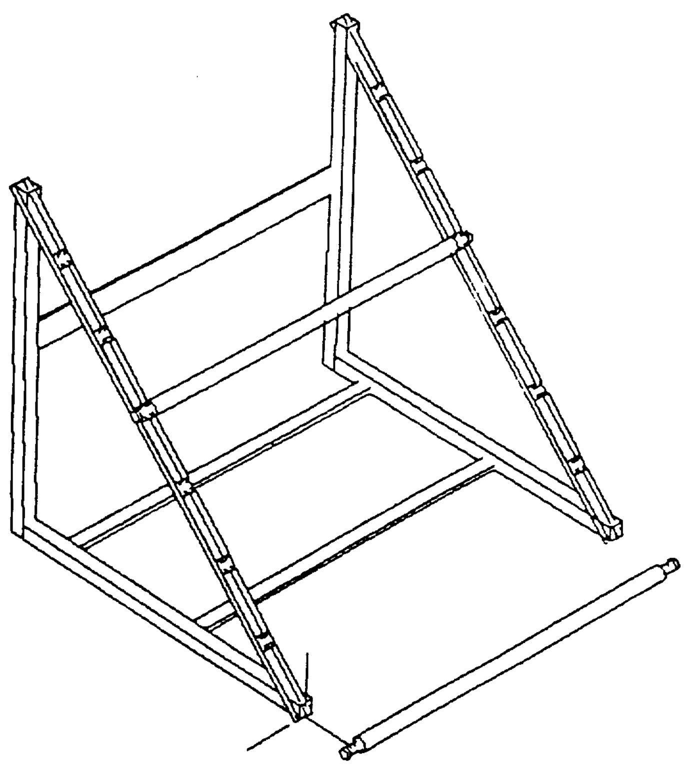


Figure B4 - Strut Rack and Strut

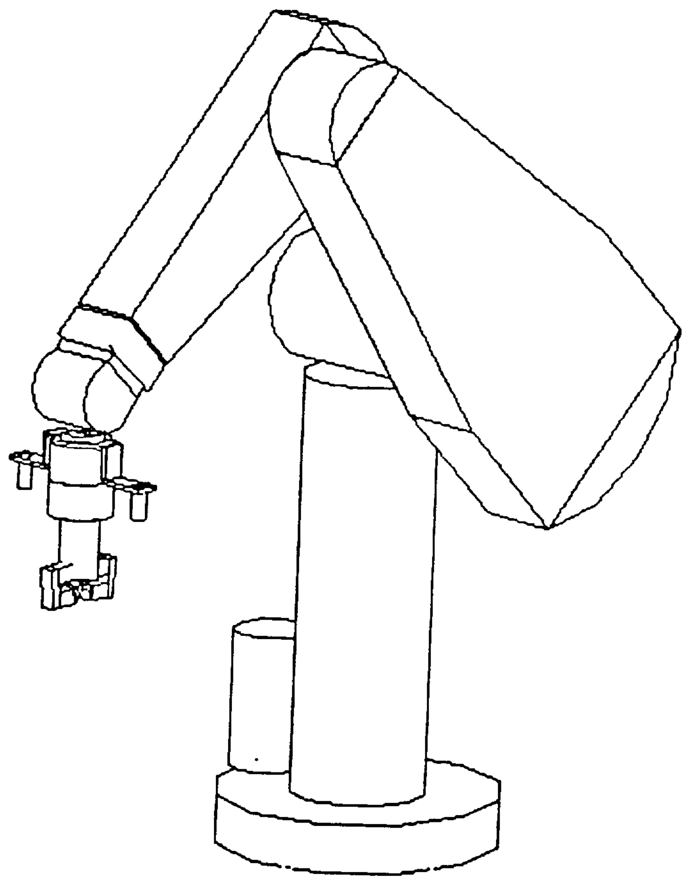


Figure B5 - PUMA 560 in Ready Position with Left Gripper

APPENDIX C

SUPPORTING CODE

This appendix contains the listed programs for all supporting code developed to interface CimStation with the MCS/CTOS system for controlling the testbed robots. The files which are contained in this appendix can be found as software in:

/home/hron/interface/

The code listed here consists of:

```
/*
**
**                       NOTICE OF COPYRIGHT
**             Copyright (C) Rensselaer Polytechnic Institute.
**                      1992 ALL RIGHTS RESERVED.
**
** Permission to use, distribute, and copy is granted ONLY for research
** purposes, provided that this notice is displayed and the author is
** acknowledged.
**
** This software was developed at the facilities of the Center for
** Intelligent Robotic Systems for Space Exploration, Troy, New York,
** thanks to generous project funding by NASA.
**
** File:          sgrincMsgHandler.c
**
** Written By:    Anna B. Hron
**
** Date:          26 OCT 1992
**
** Purpose:       MESSAGE HANDLER FOR COMMUNICATION BETWEEN CTOS AND CIMSTATION
**                VIA SOCKETS.
** Notes:
**
** To Be Done:
**
** Modification History:
*/

/*
**========== INCLUDES ==========
*/

#include "sgrinc.h"


/*
**========== PROTOTYPES ==========
*/
int newConnect(int fd);
int callHandle(int fd, TID_TYPE sgTid, ROBOT_KEY l_key, ROBOT_KEY r_key);

void move_cart(TID_TYPE sgTid, char *cs_msg, ROBOT_KEY key);
void move_puma(TID_TYPE sgTid, char *cs_msg, ROBOT_KEY key);
void move_full(TID_TYPE sgTid, char *cs_msg, ROBOT_KEY key);

void grip_it(TID_TYPE sgTid, char *cs_msg, ROBOT_KEY l_key,ROBOT_KEY r_key);
void comp_it(TID_TYPE sgTid, char *cs_msg, ROBOT_KEY l_key,ROBOT_KEY r_key);
```

```
int word_to_int(char *word);
int move_rob(TID_TYPE myTid, ROBOT_KEY key, double *jVect);


/*
**=========== grincMsgHandler ==========
*/

CtosTask(sgrincMsgHandler)

int sgrincMsgHandler(TID_TYPE sgrincTid, MSG_TYPE *msg)
{
/*----- LOCAL VAR DECLARAIONS -----*/


   int   temp_portnum;
   unsigned short   portnum;
   char *prompt_str;
   ROBOT_KEY bed_key, lf_key, rt_key;

   switch(msg->command)
   {
      case MSG_AINIT:
         recInfo(sgrincTid, "IN AINIT... \n");
      break;

      case MSG_PINIT:
         recInfo(sgrincTid, "IN PINIT... \n");

      break;

      case MSG_AEXEC:
         recInfo(sgrincTid, "IN AEXEC... \n");

         /*===== ENGAGE WHOLE BED AND GET KEY -> KEYS ONCE!!=====*/
         bed_key = clifRobotStart(sgrincTid, FULL_ARM, RIGHT_ARM);
         if (bed_key != -1) /* valid key gotten*/
{
            recInfo(sgrincTid,"18 DOF KEY = %d\n",bed_key);

/*            clifRobotSplit(bed_key, &lf_key, &rt_key);
            recInfo(sgrincTid,"GOT KEY SPLIT\n");*/

            lf_key = 33;
            recInfo(sgrincTid,"LF 9DOF KEY = %d\n",lf_key);

            recInfo(sgrincTid,"RT 9DOF KEY = %d\n",rt_key);

            prompt_str = recPrompt(sgrincTid,NULL,"ENTER portnum :");
            sscanf(prompt_str,"%d",&temp_portnum);

            portnum = (unsigned short) temp_portnum;
```

```
            sockServerCreate(sgrincTid, lf_key, rt_key, portnum, 0, 0,
                             newConnect, callHandle);
            /* TAKE ACTION IF NO CONNECT */

            msgApplicationExit(sgrincTid);
         }
         else
         {
            recInfo(sgrincTid,"BED_KEY NOT RECEIVED!!\n");
            break;
         }
      break;
   }
   return(msgDefaultProc(sgrincTid, msg));
}


/* returning 1 will cause server to exit */
int newConnect(int fd)
{
   printf("Accepted new client socket connection.\n");
   return(0);
}

/* ==================================================================
   THIS IS ACTUALLY THE MASTER FUNCTION GOVERNING INCOMING CS COMMANDS
   returning 1 will cause server to exit
   ==================================================================
*/
int callHandle(int fd, TID_TYPE sgTid, ROBOT_KEY l_key, ROBOT_KEY r_key)
{
   char cs_msg[350];
   ROBOT_KEY key;

   /* CHECK FOR VALIDITY OF MESSAGE */
   if(sockStrmRecv(fd, cs_msg, 350) < 0)
      return(1);

   recInfo(sgTid,"Recieved '%s'.\n", cs_msg);
   recInfo(sgTid,"KEYS (LF,RT): %d  %d\n",l_key, r_key);

 /*===== COMPARE MESSAGE WITH COMMAND =====*/

   /*----- IF PLAT IS TO BE MOVED -----*/
   if (strncmp(cs_msg,"C",1) == 0)
   {
      recInfo(sgTid,"RECEIVED MOVE ANGLES FOR PLAT\n");

      if (strncmp(cs_msg, "CL",2) == 0)  key = l_key;
      else   key = r_key;
```

```
      move_cart(sgTid, cs_msg, key);
   }

   /*----- IF PUMA IS TO BE MOVED -----*/
   if (strncmp(cs_msg,"P",1) == 0)
   {
      recInfo(sgTid,"RECEIVED MOVE ANGLES FOR PUMA\n");

      if (strncmp(cs_msg, "PL",2) == 0)  key = l_key;
      else   key = r_key;

      move_puma(sgTid, cs_msg, key);
   }

   /*----- IF FULL_ARM IS TO BE MOVED -----*/
   if (strncmp(cs_msg,"F",1) == 0)
   {
      recInfo(sgTid,"RECEIVED MOVE ANGLES FOR FULL\n");

      if (strncmp(cs_msg, "FL",2) == 0)  key = l_key;
      else  key = r_key;

      move_full(sgTid, cs_msg, key);
   }

   /*----- OPERATE GRIPPERS -----*/
   if (strncmp(cs_msg,"G",1) == 0)
      grip_it(sgTid, cs_msg, l_key, r_key);

   /*----- (DIS)ENGAGE COMPLIANCE -----*/
   if (strncmp(cs_msg,"C",1) == 0)
      comp_it(sgTid, cs_msg, l_key, r_key);

   /*----- THE ONLY COMMAND THAT TERMINATES SOCKET CONNECTION -----*/
   if (cs_msg[0] == 'q')
   {
      clifRobotEnd(sgTid,l_key);
      clifRobotEnd(sgTid,r_key);
      return(1);
   }
   else
      return(0); /* RETURNS TO THE BEGINNING OF THE LOOP - NO EXIT! */
}

/* ===========================
   PROCEDURE TO MOVE A PLATFORM
   ===========================
*/
void move_cart(TID_TYPE sgTid,char *cs_msg,ROBOT_KEY key)
{
   int jts, i, j;
   char dummy, word[9][32];
```

```
    double curr_jvect[9], jVect[9], temp;

    recInfo(sgTid,"INSIDE MOVE CART\n");
    recInfo(sgTid,"KEY C: %d \n",key);

 /*   if ((clifRobotWhere(sgTid, key, curr_jvect)) == OK)
    {
       recInfo(sgTid,"CR WHERE C= curr_jv: %f %f %f %f %f %f %f %f %f \n",
            curr_jvect[0], curr_jvect[1], curr_jvect[2], curr_jvect[3],
            curr_jvect[4], curr_jvect[5], curr_jvect[6], curr_jvect[7],
            curr_jvect[8]);
 */

       if ((jts = sscanf(cs_msg,"%c %s %s %s", &dummy, word[0], word[1],
                                             word[2])) != 4)
          recInfo(sgTid,"NO. JTS INCOMPATIBLE WITH ARM ENGAGED!\n");

       jVect[0] = (word_to_int(word[0])) * (.00001); /* LINEAR JOINT */
       for (i=1; i<3; i++)
       {
          temp = (word_to_int(word[i])) * (.001);
          jVect[i]  = DEG_TO_RAD(temp);

          recInfo(sgTid,"Defined 1-3 of jVect= %f %f %f\n",jVect[0],
                         jVect[1],jVect[2]);
       }

       jVect[3] = 0.0;  jVect[4] = -0.785;  jVect[5] = 3.141;
       jVect[6] = 0.0;  jVect[7] = 0.785;   jVect[8] = 1.571;

 /*      for (j=3; j<9; j++)  SET JTS 4-9 TO VALUE BEFORE MOVE ie NO CHANGE
       {
          jVect[j] = curr_jvect[j];
       }
 */

       recInfo(sgTid,"MOVING TO JVECT: %f %f %f %f %f %f %f %f %f \n",jVect[0],
                jVect[1], jVect[2], jVect[3], jVect[4], jVect[5], jVect[6],
                jVect[7], jVect[8]);

       if (move_rob(sgTid, key, jVect) != OK)
          recInfo(sgTid,"PLAT MOVE UNSUCCESSFUL...\n");
 /*   }
    else
       recInfo(sgTid,"UNABLE TO QUERY PLAT FOR POSITION\n");
 */
 }


 /* =======================
    PROCEDURE TO MOVE A PUMA
    =======================
```

```
*/
void move_puma(TID_TYPE sgTid,char *cs_msg,ROBOT_KEY key)
{
   int jts, i, j;
   char dummy, word[9][32];
   double curr_jvect[9], jVect[9], temp;

   recInfo(sgTid,"INSIDE MOVE PUMA\n");
   recInfo(sgTid,"KEY P: %d \n",key);

/*   if (clifRobotWhere(sgTid, key, curr_jvect) == OK)
   {
      recInfo(sgTid,"CR WHERE P= curr_jv: %f %f %f %f %f %f %f %f %f \n",
           curr_jvect[0], curr_jvect[1], curr_jvect[2], curr_jvect[3],
           curr_jvect[4], curr_jvect[5], curr_jvect[6], curr_jvect[7],
           curr_jvect[8]);
*/

      if ((jts = sscanf(cs_msg,"%c %s %s %s %s %s %s",&dummy, word[3], word[4],
                   word[5], word[6], word[7], word[8])) != 7)
         recInfo(sgTid,"NO. JTS INCOMPATIBLE WITH ARM ENGAGED!\n");

      for (i=3; i<9; i++)
      {
         recInfo(sgTid,"i = %d\n",i);
         temp = (word_to_int(word[i])) * (.001);
         jVect[i]  = DEG_TO_RAD(temp);

         recInfo(sgTid,"jVect 1-6 = %f %f %f %f %f %f\n", jVect[3], jVect[4],
                 jVect[5], jVect[6], jVect[7], jVect[8]);
      }

      jVect[0] = 1.3;  jVect[1] = 0.0;  jVect[2] = 0.0;

/*      for (j=0; j<3; j++) SET JTS 1-3 TO VALUE BEFORE MOVE ie NO CHANGE
      {
         jVect[j] = curr_jvect[j];
      }
*/
      recInfo(sgTid,"MOVING TO JVECT: %f %f %f %f %f %f %f %f %f \n",jVect[0],
              jVect[1], jVect[2], jVect[3], jVect[4], jVect[5], jVect[6],
              jVect[7], jVect[8]);

      if (move_rob(sgTid, key, jVect) != OK)
         recInfo(sgTid,"PUMA MOVE UNSUCCESSFUL...\n");
/*    }
   else
      recInfo(sgTid,"UNABLE TO QUERRY PUMA FOR POSITION\n");
*/
}
```

```
/* ==========================
   PROCEDURE TO MOVE A FULL ARM
   ==========================
*/
void move_full(TID_TYPE sgTid,char *cs_msg,ROBOT_KEY key)
{
   int jts, i;
   char dummy, word[9][32];
   double jVect[9], temp;


   recInfo(sgTid,"INSIDE MOVE FULL ARM\n");
   recInfo(sgTid,"KEY  F: %d \n",key);

   if ((jts= sscanf(cs_msg,"%c %s %s %s %s %s %s %s %s %s",&dummy, word[0],
        word[1], word[2], word[3], word[4], word[5], word[6], word[7],
        word[8])) != 10)
      recInfo(sgTid,"NO. JTS INCOMPATIBLE WITH ARM ENGAGED!\n");

   jVect[0] = (word_to_int(word[0])) * (.00001); /* LINEAR JOINT */
   for (i=1; i<9; i++)
   {
      temp = (word_to_int(word[i])) * (.001);
      jVect[i]  = DEG_TO_RAD(temp);

      recInfo(sgTid,"jVect 1-9 = %f %f %f %f %f %f %f %f %f\n", jVect[0],
                     jVect[1], jVect[2], jVect[3], jVect[4], jVect[5],
                     jVect[6], jVect[7], jVect[8]);
   }
   recInfo(sgTid,"MOVING TO JVECT: %f %f %f %f %f %f %f %f %f \n",jVect[0],
                  jVect[1], jVect[2], jVect[3], jVect[4], jVect[5], jVect[6],
                  jVect[7], jVect[8]);

   if (move_rob(sgTid, key, jVect) != OK)
      recInfo(sgTid,"FULL MOVE UNSUCCESSFUL...\n");
}


/* ====================================================
   Procedure TO FIND ROBOT POSITION AND SEND TO CIMSTATION
   ====================================================

void curr_pos(TID_TYPE sgTid, char *cs_msg, ROBOT_KEY key, double *curr_pos_jv)
{
   recInfo(sgTid,"INSIDE CURR POS \n");

   clifRobotWhere(sgTid,key,*curr_pos_jv);
*/


/* ==========================
   PROCEDURE TO OPERATE GRIPPER
```

```
   ============================
*/
void grip_it(TID_TYPE sgTid, char *cs_msg, ROBOT_KEY l_key,ROBOT_KEY r_key)
{
   char cs_msg_temp[350];
   ROBOT_KEY key;
   ARM_TYPE type_arm;

   recInfo(sgTid,"INSIDE GRIPPER \n");
   recInfo(sgTid,"KEYS (LF,RT): %d  %d\n",l_key, r_key);

   strcpy(cs_msg_temp,cs_msg);  /* FOR SAFE KEEPING */

   if (strchr(cs_msg_temp,'L') != NULL)
     {
        key = l_key;
        type_arm = LEFT_ARM;
     }
   else
     {
        key = r_key;
        type_arm = RIGHT_ARM;
     }

   if (strncmp(cs_msg,"G O",3) == 0)
   {
      clifGripperOpen(sgTid,key,type_arm,CLIF_NOWAIT);
      recInfo(sgTid,"OPENED GRIPPER");
   }
   else if (strncmp(cs_msg,"G C",3) == 0)
   {
      clifGripperClose(sgTid,key,type_arm,CLIF_NOWAIT);
      recInfo(sgTid,"CLOSED GRIPPER");
   }
   return;
}


/* ===================================
   PROCEDURE TO (DIS)ENGAGE COMPLIANCE
   ===================================
*/
void comp_it(TID_TYPE sgTid, char *cs_msg, ROBOT_KEY l_key,ROBOT_KEY r_key)
{
   char cs_msg_temp[350];
   ROBOT_KEY key;
   ARM_TYPE type_arm;

   recInfo(sgTid,"INSIDE COMP_IT \n");
   recInfo(sgTid,"KEYS (LF,RT): %d  %d\n",l_key, r_key);

   strcpy(cs_msg_temp,cs_msg);  /* FOR SAFE KEEPING */
```

```
    if (strchr(cs_msg_temp,'L') != NULL)
      {
         key = l_key;
         type_arm = LEFT_ARM;
      }
    else
      {
         key = r_key;
         type_arm = RIGHT_ARM;
      }

    if (strncmp(cs_msg,"C O",3) == 0)
    {
       if (clifComplianceOn(sgTid,key,type_arm,CLIF_NOWAIT) == OK)
          recInfo(sgTid,"TURNED COMPLIANCE ON");
       else
          recInfo(sgTid,"UNABLE TO ACTIVATE COMPLIANCE...\n");
    }
    else if (strncmp(cs_msg,"C F",3) == 0)
    {
       if (clifComplianceOff(sgTid,key,type_arm) == OK)
          recInfo(sgTid,"TURNED COMPLIANCE OFF");
       else
          recInfo(sgTid,"UNABLE TO ACTIVATE COMPLIANCE...\n");
    }
    return;
}


/* ================================================
   FUNCTION TO TRANSLATE WORDS (32 BITS) INTO AN INT
   ================================================
*/
int word_to_int(char *word)
{
   int power, sum, i;

   for (sum=0, power=1, i=0; i<32; i++, power <<=1)
      if (word[31-i] == '1')
         sum += power;

   return(sum);
}


/* ============================================
   THIS IS THE moveHome ROUTINE FROM THE CLIF EX.
   ============================================
*/
int move_rob(TID_TYPE myTid, ROBOT_KEY key, double *jVect)
{
```

```
  ROBOT_MODE *mode = NULL;
  ROBOT_KNOTPT *knotpt = NULL;


   recInfo(myTid,"MOVE KEY: %d \n",key);

  /* set up a mode with a speed of 0.3 and a blend of 0.5 */
  mode = clifModeSet(NULL,0.3,0.0,0.5,0.5);

  recInfo(myTid,"SETTING MODE\n");

  /* make sure that the clifModeSet succeeded */
  if (mode == NULL)
  {
    recError(myTid,"Unable to allocate mode structure.\n");
    return(ERROR);
  }

  knotpt = clifKnotptSet(NULL, JOINT9, jVect);
  recInfo(myTid,"SETTING KNOTPT\n");

  /* make sure the clifKnotptSet succeeded */
  if (knotpt == NULL)
  {
    recError(myTid,"Unable to allocate knotpt structure.\n");
    clifModeRelease(mode);
    return(ERROR);
  }

  /* move the robot with the appropriate key, mode and knotpt */
  if (clifRobotMove(myTid,key,mode,knotpt,CLIF_WAIT) != OK)
    recError(myTid,"Move failed.\n");

  recInfo(myTid,"MOVING ROBOT TO jVect: %f %f %f %f %f %f %f %f %f\n",
          jVect[0], jVect[1], jVect[2], jVect[3], jVect[4], jVect[5],
          jVect[6], jVect[7], jVect[8]);

/* release mode and knotpt structures */
  clifModeRelease(mode);
  clifKnotptRelease(knotpt);

  return(OK);
}
```

```c
/*
**
**                       NOTICE OF COPYRIGHT
**            Copyright (C) Rensselaer Polytechnic Institute.
**                      1992 ALL RIGHTS RESERVED.
**
** Permission to use, distribute, and copy is granted ONLY for research
** purposes, provided that this notice is displayed and the author is
** acknowledged.
**
** This software was developed at the facilities of the Center for
** Intelligent Robotic Systems for Space Exploration, Troy, New York,
** thanks to generous project funding by NASA.
**
** File:        sgrinc.h
**
** Written By:  Anna B. Hron
**
** Date:        26 OCT 1992
**
** Purpose:     HEADER FILE FOR sgrincMsgHandler routine.
**
** Notes:
**
** To Be Done:
**
** Modification History:
*/


#ifndef INCsgrincH
#define INCsgrincH

/*========== INCLUDE FILES ==========*/
#include <stdio.h>
#include <stdlib.h>
#include <msgLib.h>
#include <recLib.h>
#include <mcsLib.h>
#include <btsLib.h>
#include <ctos.h>
#include <string.h>

#include "clif.h"

#include <unistd.h>
#include <errno.h>
#include <netdb.h>
```

```
/* INCLUSION OF sockLib.h */

#ifndef INCsockLibH
#define INCsockLibH

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MAX_HOST_NAME_LENGTH   128


/*========== SOCKET FUNCTION PROTOTYPES ==========*/
typedef int SOCKET_TYPE;


void sockServerCreate(TID_TYPE sgrincTid, ROBOT_KEY lf_key, ROBOT_KEY rt_key,
                        unsigned short portnum, long sec,
                        long usec, int (*acceptFunc)(int fd),
                        int (*callFunc)(int fd, TID_TYPE sgTid,
                        ROBOT_KEY l_key, ROBOT_KEY r_key));

int sockEstablish(const unsigned short portnum, struct sockaddr_in *sa,
  const SOCKET_TYPE sockType);
int sockConnectAccept(int s);

int sockStrmRecv(int s, register char *data, unsigned dataSize);

#endif INCsockLibH


/*========== GRINC FUNCTION PROTOTYPES ==========*/

int  sgrincMsgHandler (TID_TYPE myTid, MSG_TYPE *msg);

#endif INCsgrincH
```

```
/*
**
** File:        sgrincLib.c
**
** Written By:  Anna B. Hron (Keith Nicewarner)
**
** Date:        26 OCT 1992
**
** Purpose:     Common stream socket routines.
**
** Modification History:  MODIFIED TO SUIT OPERATIONS OF GRINC.
*/

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>

#include "sgrinc.h"


/* create simple stream socket monitor for a *single* client */
void sockServerCreate(TID_TYPE sgrincTid, ROBOT_KEY lf_key, ROBOT_KEY rt_key,
                      unsigned short portnum, long sec,
                      long usec, int (*acceptFunc)(int fd),
                      int (*callFunc)(int fd, TID_TYPE sgTid, ROBOT_KEY l_key,
                      ROBOT_KEY r_key))
{
   int fdServer;
   int fdClient = -1;
   struct sockaddr_in addrServer;
   fd_set readfds;
   struct timeval timeOut;
   struct timeval *timeout;
   int numPending;

   if((fdServer = sockEstablish(portnum, &addrServer, SOCK_STREAM)) < 0)
   {
      perror("establishing stream socket");
      return;
   }
   listen(fdServer, 3);

   printf("\nSimple stream socket server running.\n");

   if((sec == 0) && (usec == 0))
      timeout = NULL;
```

```
    else
    {
       timeOut.tv_sec  = sec;
       timeOut.tv_usec = usec;
       timeout = &timeOut;
    }
    while(1)
    {
       FD_ZERO(&readfds);
       FD_SET(fdServer, &readfds);
       if(fdClient != -1)
  FD_SET(fdClient, &readfds);

       /* block until message pending or timeout */
       while(((numPending =
       select(getdtablesize(), &readfds,NULL,NULL, timeout)) < 0)
               && (errno == EINTR));

       if(numPending == 0)
       {
  if(timeout != NULL)
     printf("Server timed out after %ld seconds,"
    "%ld microseconds of no client calls.\n",
    timeout->tv_sec, timeout->tv_usec);
  break;
       }

       if(FD_ISSET(fdServer, &readfds))
       {
          if((fdClient = sockConnectAccept(fdServer)) == -1)
  {
     perror("accepting client connection");
     break;
  }
  else if((acceptFunc != NULL) && acceptFunc(fdClient))
     break;
       }
       if((fdClient != -1) && FD_ISSET(fdClient, &readfds))

          /* LOOP ONLY TERMINATES WHEN MESSAGE IS 'q'*/
  if((callFunc != NULL) && callFunc(fdClient, sgrincTid, lf_key,
                                            rt_key))
    break;
   }
   close(fdServer);
   if(fdClient != -1)
      close(fdClient);
}


/*
** Establish a stream socket in a given port,
```

```
** return address and file descriptor.
*/
int sockEstablish(const unsigned short portnum, struct sockaddr_in *sa,
  const SOCKET_TYPE sockType)
{
   char myname[MAX_HOST_NAME_LENGTH+1];
   int s;
   struct hostent *hp;

   bzero(sa, sizeof(struct sockaddr_in));
   gethostname(myname, MAX_HOST_NAME_LENGTH);
   hp = gethostbyname(myname);
   if(hp == NULL)
      return(-1);
   sa->sin_family = hp->h_addrtype;
   sa->sin_port = htons(portnum);
   if((s = socket(AF_INET, sockType, 0)) < 0)
      return(-1);
   if(bind(s, (struct sockaddr *)sa, sizeof(struct sockaddr_in)) < 0)
   {
      close(s);
      return(-1);
   }
   return(s);
}

/* accept a new socket connect to a server */
int sockConnectAccept(int s)
{
   struct sockaddr_in sa;
   int len;

   len = sizeof(sa);
   getsockname(s, (struct sockaddr *)&sa, &len);
   return(accept(s, (struct sockaddr *)&sa, &len));
}

/* THIS WILL BE ADDED WHEN TWO-WAY COMMUNICATION IS ESTABLISHED
   connect to an existing socket on a given port on a given host
int sockConnect(const char *hostname, const unsigned short portnum,
struct sockaddr_in *sa, const SOCKET_TYPE sockType)
{
   struct hostent *hp;
   int s;

   if((hp = gethostbyname(hostname)) == NULL)
   {
      errno = ECONNREFUSED;
      printf("Unknown host '%s'.\n", hostname);
      return(-1);
   }
```

```
    bzero(sa, sizeof(struct sockaddr_in));
    bcopy(hp->h_addr, (char *)&sa->sin_addr, hp->h_length);
    sa->sin_family = hp->h_addrtype;
    sa->sin_port = htons((u_short)portnum);

    if((s = socket(hp->h_addrtype, sockType, 0)) < 0)
    {
       perror("Error getting server stream socket");
       return(-1);
    }
    if(connect(s, (struct sockaddr *)sa, sizeof(struct sockaddr_in)) < 0)
    {
       close(s);
       perror("Error connecting to server stream socket");
       return(-1);
    }
    return(s);
}*/


/*
** Stream socket I/O routines
*/

/* THIS WILL BE ADDED WHEN TWO-WAY COMMUNICATION IS ESTABLISHED
 write data to stream socket
int sockStrmSend(int s, register char *data, unsigned dataSize)
{
   register int bcount = 0;
   register int br;

   while(bcount < dataSize)
   {
      if((br = write(s, data, dataSize - bcount)) > 0)
      {
         bcount += br;
         data += br;
      }
      else if(br < 0)
      {
         perror("sockStrmSend");
         return(br);
      }
   }
   return(bcount);
}*/

/* read data from stream socket */
int sockStrmRecv(int s, register char *data, unsigned dataSize)
{
   register int bcount = 0; /* counts bytes read */
   register int br;         /* bytes read this pass */
```

```
   while(bcount < dataSize)
   {
      if((br = read(s, data, dataSize - bcount)) > 0)
      {
         bcount += br;
         data += br;
      }
      else if(br < 0)
      {
         perror("sockStrmRecv");
         return(br);
      }
   }
   return(bcount);
}
```

```
/*  IMAKEFILE FOR GRINC APPLICATION */

/*===== SERARCH DIRECTORIES AND LOAD LIBRARIES =====*/

LDLIBS += -lrec
LDLIBS += -lctos
LDLIBS += -lmsg
LDLIBS += -lbts
LDLIBS += -lmcs

LDLIBS += -lclifClient

LDLIBS += -lkntpt
LDLIBS += -lconfig

LDLIBS += -ltrans
LDLIBS += -lm

AllTarget(sgrincMsgHandler )

UNIXBinTarget(sgrincMsgHandler, sgrincMsgHandler.o sgrincLib.o )
```

```
% CTOS CONFIGURATION  FILE FOR GRINC ON MERCURY

% File:         sgrinc.cfg
% Written by:   Anna Hron
% Date:         30 Oct 1992
%
% Purpose:      Boots the MCS for running the GRaphical INterface between
%               Cimstation and the CIRSSE testbed.
% Mod History: Modified to communicate through sockets
% Notes:        Application is on pluto - (may later be changed)


%== TASKS CPUS 0-5 WITH APPROPRIATE CODE MSGHDLRS ==

chassis vx0 1  /usr2/testbed/exp/vxworks/demos/clif/clif.cfg_fast
chassis mercury

sequencer mercury

PREFIX mercury 0


%===== LOAD CODE =====

%0 echo LOADING GRINC...
chdir /home/hron/ctos/testbed/grinc
task sgrinctask sgrincMsgHandler 111
```

```
{ File:       testbed_menu.sil
  Written by: Anna Hron
  Date:       13 Oct1992

  Purpose:    This file contains SIL code for the Cimstation/Testbed
              menu interface.
  Modifications:  This menu will interface with the CTOS message Handler
                  (and Clif) via sockets.
                  Also there will be a two-way communication, as well as
                  several other improvements. -ABH 14 OCT 92-
}

var
     file_d : integer;
     host_name : string;
     port_n : integer;
     oj_last3 : darray of real;
     old_jts, old_jts_init : jv;
     sum : real;

     comp, grip : string;

     curr_cell,curr_rob,curr_rob_msg : string;
     eng_rob, eng_rob_msg : string;

     move_3dof, move_6dof, move_9dof : jv;

     last_3dof, l3_zero : darray of real;
     fl_ready,fr_ready,cl_ready,cr_ready,puma_ready  : jv;
     fl_zero,fr_zero,cl_zero,cr_zero,puma_zero  : jv;;

{===== GENERAL DECLARATIONS =====}
file_d := 0;
host_name := 'none';
port_n := 0;
oj_last3 == array_create(real,2);
oj_last3[0]:=0.0;  oj_last3[1]:=0.0;  oj_last3[2]:=0.0;
old_jts == [0,0,0,0,0,0,oj_last3] as jv;
old_jts_init == [0,0,0,0,0,0,oj_last3] as jv;

comp := 'not set';
grip := 'not set';

curr_rob := 'none';
curr_cell := 'none';
curr_rob_msg := 'none';

last_3dof == array_create(real,2);
```

```
l3_zero == array_create(real,2);

{----- CIRSSE READY POSITIONS -----}
last_3dof[0]:=0.0;  last_3dof[1]:=45.0;  last_3dof[2]:=90.0;
fl_ready == [-130,0,0,0,-45,180,last_3dof] as jv;
fr_ready == [130,0,0,0,-45,180,last_3dof] as jv;

cl_ready == [-130,0,0,0,0,0] as jv;
cr_ready == [130,0,0,0,0,0] as jv;
puma_ready == [0,-45,180,0,45,90] as jv;

{----- PUMA ZERO POSITIONS -----}
l3_zero[0]:=0.0;  l3_zero[1]:=.0;  l3_zero[2]:=0.0;
fl_zero == [-130,0,0,0,0,0,l3_zero] as jv;
fr_zero == [130,0,0,0,0,0,l3_zero] as jv;

cl_zero == [-130,0,0,0,0,0] as jv;
cr_zero == [130,0,0,0,0,0] as jv;
zero_puma == [0,0,0,0,0,0] as jv;


{----- REMOVE EVERYTHING FROM THE WORLD -----}
set_up_new_world();


{===== MENU DEFINITIONS =====}
{---------- TOP MENU ----------}
testbed_menu == mk_imenu('TESTBED', ulist('SOCKET CONNECT','CELL',(list(
  'CURR ROBOT','!INFO!')),'HOME',list('MOTION','OFF'),'GRIPPER',(list(
  'COMPLIANCE','!INFO!')),'CURRENT POSITION','END SESSION'),
  "testbed_menu_handler);

set_msg(testbed_menu,'CURR ROBOT',curr_rob);
set_msg(testbed_menu,'COMPLIANCE',comp);

{---------- SUB MENUS ----------}
connect_menu == mk_imenu('SOCKET CONNECT',ulist((list('HOSTNAME','!INFO!')),
  (list('PORTNUMBER','!INFO!')),'CONNECT - OK'),"connect_menu_handler);

set_msg(connect_menu,'HOSTNAME',host_name);
set_msg(connect_menu,'PORTNUMBER',port_n);

cell_menu == mk_imenu('CELLS',ulist('2x9DOF ARMS','2x6DOF+2x3DOF ARMS'),
  "cell_menu_handler);

home_menu == mk_imenu('HOME',ulist('CURRENT TO HOME','ALL TO HOME',
  'CURRENT TO ZERO','ALL TO ZERO'),"home_menu_handler);

motion_menu == mk_imenu('MOTION',ulist('TO GOAL','VIA PATH','----->',
  'PREVIEW/CREATE','VIEW STORED PATH','ON','OFF'),"motion_menu_handler);

gripper_menu == mk_imenu('GRIPPER',ulist(list('CURR ROBOT','!INFO!'),'OPEN',
```

```
  'CLOSE'),"gripper_menu_handler);

{===== DEFINE FUNCTIONS FOR EACH MENU SELECTION =====}
{---------- TOP MENU ----------}

procedure current_rob;
var
     crob_anno : anno_string;
begin
     set_msg(testbed_menu,'CURR ROBOT',curr_rob);

     if (curr_cell = '2x9') then
        begin
           write_msg('SELECT A ROBOT TO WORK WITH : [FL, FR]:');
           crob_anno := get_a_string();
           if ((crob_anno.val <> 'FL') and (crob_anno.val <> 'FR')) then
              begin
                 activate(testbed_menu);
                 write_msg('ROBOT SELECTION NOT VALID - TRY AGAIN');
              end
           else
              begin
                 curr_rob := crob_anno.val;
                 if (curr_rob = 'FL') then curr_rob_msg := 'FULL LEFT ARM';
                 if (curr_rob = 'FR') then curr_rob_msg := 'FULL RIGHT ARM';
                 write_msg(concat(curr_rob_msg,' IS THE CURRENT ROBOT'));
              end;
        end;

     if (curr_cell = '2x6+2x3') then
        begin
           write_msg('SELECT A ROBOT TO WORK WITH : [PL, PR, CL, CR]:');
           crob_anno := get_a_string();
           if ((crob_anno.val <> 'PL') and (crob_anno.val <> 'PR') and
               (crob_anno.val <> 'CL') and (crob_anno.val <> 'CR')) then
              begin
                 activate(testbed_menu);
                 write_msg('ROBOT SELECTION NOT VALID - TRY AGAIN');
              end
           else
              begin
                 curr_rob := crob_anno.val;
                 if (curr_rob = 'PL') then curr_rob_msg := 'LEFT PUMA';
                 if (curr_rob = 'PR') then curr_rob_msg := 'RIGHT PUMA';
                 if (curr_rob = 'CL') then curr_rob_msg := 'LEFT PLATFORM';
                 if (curr_rob = 'CR') then curr_rob_msg := 'RIGHT PLATFORM';
                 write_msg(concat(curr_rob_msg,' IS THE CURRENT ROBOT'));
              end;
        end;

     if ((curr_cell <> '2x9') and (curr_cell <> '2x6+2x3')) then
        begin
```

```
            activate(testbed_menu);
            write_msg('SELECT A WORKCELL BEFORE TRYING TO ENGAGE A ROBOT');
         end;
 end;  {---------- END current_rob ----------}


 procedure comp_o_f;
 var
      comp_sure : anno_string;
 begin
      if ((comp = 'not set') or (comp = 'OFF')) then
      begin
         write_msg('DO YOU WANT COMPLIANCE OF THE **CURRENT** ARM ON? (y|n)');
         comp_sure := get_a_string();
         if (comp_sure.val = 'y') then
         begin
            comp := 'ON';
            sil_sock1_data(file_d,('C O '*curr_rob));
            writeln('SENT COMPLIANCE ON THROUGH SOCKET');
         end;
      end

      else {COMPLIANCE = ON}
      begin
         write_msg('DO YOU WANT COMPLIANCE OF THE **CURRENT** ARM OFF? (y|n)');
         comp_sure := get_a_string();
         if ((comp_sure.val = 'y') or (comp_sure.val = 'Y')) then
         begin
            comp := 'OFF';
            sil_sock1_data(file_d,('C F '*curr_rob));
            writeln('SENT COMPLIANCE OFF THROUGH SOCKET');
         end;
      end;
 end;


 procedure currentPosition;
 begin
      write_msg('WILL QUERY TESTBED CURRENT POSITION, AND UPDATE CIMSTATION');
 {
      sil_sock1_data(file_d, ('W '*curr_rob));
      sil_sock_recv(file_d, recxjv);
      writeln(rexjv);
 }
 end;  {---------- END CheckPosition ----------}


 procedure sendingOff;
 var
      k : integer;
 begin
      refresh_actions := cdr(refresh_actions);
```

```
    old_jts := old_jts_init;

    write_msg('OFF = TESTBED ARMS NO LONGER RESPONDING TO CIMSTATION MOVES.');
end;    {---------- END sendingOff ----------}



{===== SUPPORTING FUNCTIONS AND PROCEDURES =====}

process cell_2bots();
begin
    refresh(0.0);
    { TRY TO FIND A WAY TO DO A REMOVE ALL HERE }
    install('FL','left_9dof');
    moveto('FL',fl_ready);
    install('left_gripper','/home/hron/silma/tools/left_gripper_o.ee');
    mount('left_gripper','FL');
    install('FR','right_9dof');
    moveto('FR',fr_ready);
    install('right_gripper','/home/hron/silma/tools/right_gripper_a.ee');
    mount('right_gripper','FR');

    {----- COLORS TO MATCH ACTUAL TESTBED -----}
    paint('FL',cyan); paint('FL/link0',gray);
    paint('FL/link3/link0',white);paint('FL/link4',white);
    paint('FL/link5',white); paint('FL/link6',white);
    paint('FL/link7',white);paint('FL/link8',white);
    paint('FL/linkn',white);
    paint('left_gripper',silver); paint('left_gripper/fts',ivory);
    paint('FR',cyan); paint('FR/link0',gray);
    paint('FR/link3',white); paint('FR/link4',white);
    paint('FR/link5',white); paint('FR/link6',white);
    paint('FR/link7',white); paint('FR/link8',white);
    paint('FR/linkn',white);
    paint('right_gripper',silver); paint('right_gripper/fts',ivory);

    hide('teacher');
    view_all();
end;

process cell_4bots();
begin
    refresh(0.0);
    { TRY TO FIND A WAY TO DO A REMOVE ALL HERE }
    install('CL','left_half_plat');
    moveto('CL',cl_ready);
    install('PL','c_puma');
    moveto('PL',puma_ready); {CIRSSE READY}
    moveto('PL',pose_of('CL_flange'));
    affix('PL','CL_flange');
    install('left_gripper','/home/hron/silma/tools/left_gripper_o.ee');
    mount('left_gripper','PL');
```

```
     paint('left_gripper',silver); paint('left_gripper/fts',ivory);
     install('CR','right_half_plat');
     moveto('CR',cr_ready);
     install('PR','c_puma');
     moveto('PR',puma_ready);  {CIRSSE READY}
     moveto('PR',pose_of('CR_flange'));
     affix('PR','CR_flange');
     install('right_gripper','/home/hron/silma/tools/right_gripper_a.ee');
     mount('right_gripper','PR');
     paint('right_gripper',silver); paint('right_gripper/fts',ivory);

     hide('teacher');
     view_all();
end; {-------------------------------------------------------------}


process all_to(where:string);
var
     fl_w,fr_w, puma_w : jv;
begin
     wait(where);
     if (where = 'ready') then
        begin
           fl_w := fl_ready; fr_w := fr_ready;
           puma_w := puma_ready;
        end
     else
        begin
           fl_w := fl_zero;  fr_w := fr_zero;
           puma_w := puma_zero;
        end;

     if (curr_cell = '2x9') then
     begin
        moveto('FL',fl_w);
        moveto('FR',fr_w);
     end;

     if (curr_cell = '2x6+2x3') then
     begin
        unaffix('PL','CL_flange');
        unaffix('PR','CR_flange');
        moveto('PL',puma_w);
        moveto('PR',puma_w);

        affix('PL','CL_flange');
        affix('PR','CR_flange');

        moveto('CL',1,-130.0);  moveto('CL',2,0.0);  moveto('CL',3,0.0);
        moveto('CR',1,130.0);  moveto('CR',2,0.0);  moveto('CR',3,0.0);
     end;
```

```
      if ((curr_cell <> '2x9') and (curr_cell <> '2x6+2x3')) then
         begin
            activate(testbed_menu);
            write_msg('SELECT A WORKCELL BEFORE TRYING TO READY A ROBOT');
         end;
  end; {----- END all_to -----}


  process one_to(where:string);
  var
       where_jv : jv;
  begin
       wait(where);
       if (where = 'ready') then
          begin
             if (curr_rob = 'FL') then where_jv := fl_ready;
             if (curr_rob = 'FR') then where_jv := fr_ready;
             if (curr_rob = 'PL') then where_jv := puma_ready;
             if (curr_rob = 'PR') then where_jv := puma_ready;
             if (curr_rob = 'CL') then where_jv := cl_ready;
             if (curr_rob = 'CR') then where_jv := cr_ready;
          end
       else
          begin
             if (curr_rob = 'FL') then where_jv := fl_zero;
             if (curr_rob = 'FR') then where_jv := fr_zero;
             if (curr_rob = 'PL') then where_jv := puma_zero;
             if (curr_rob = 'PR') then where_jv := puma_zero;
             if (curr_rob = 'CL') then where_jv := cl_zero;
             if (curr_rob = 'CR') then where_jv := cr_zero;
          end;

       if (curr_cell = '2x6+2x3') then
       begin
          unaffix('PL','CL_flange');
          unaffix('PR','CR_flange');
       end;

       moveto(curr_rob,where_jv);

       if (curr_cell = '2x6+2x3') then
       begin
          moveto('PL',pose_of('CL_flange'));
          affix('PL','CL_flange');

          moveto('PR',pose_of('CR_flange'));
          affix('PR','CR_flange');
       end;
  end;  {------------------------ END one_to ------------------------}

  process send_all_to_r();
  begin
```

```
     signal(all_to,where,'ready'); end;

process send_all_to_z();
begin
     signal(all_to,where,'zero'); end;

process send_one_to_r();
begin
     signal(one_to,where,'ready'); end;

process send_one_to_z();
begin
     signal(one_to,where,'zero'); end; {----- END home_menu processes -----}


process gripper_o();
begin
     if ((curr_rob = 'PL') or (curr_rob = 'FL')) then
        open_ee('left_gripper')
     else
     begin
        if ((curr_rob = 'PR') or (curr_rob = 'FR')) then
           open_ee('right_gripper')
        else
           write_msg('NO GRIPPER COMPATIBLE WITH CURRENT ROBOT');
     end;
end;

process gripper_c();
begin
     if ((curr_rob = 'PL') or (curr_rob = 'FL')) then
        close_ee('left_gripper')
     else
     begin
        if ((curr_rob = 'PR') or (curr_rob = 'FR')) then
           close_ee('right_gripper')
        else
           write_msg('NO GRIPPER COMPATIBLE WITH CURRENT ROBOT');
     end;
end;



procedure print_cjv_msg();
{This procedure when added as a closure to the refresh_actions function,
 will write the current joint vector of the specified robot, to a data file.}

var
     j : integer;
     jt_str : darray of string;
     str_send : string;
     mse : real;
```

```
begin
     delay(1.0);

{********* AT EVERY SCREEN REFRESH, INSTEAD OF WRITELN(...) THERE WILL BE A
              SOCKSTRMSEND MESSAGE  - ROBOT + JV ****************}

     if ((curr_rob = 'CR') or (curr_rob = 'CL')) then
     begin
        jt_str := array_create(string,2);
        move_3dof := c_jv(curr_rob);

        for j := 0 to 2 do
        begin
           mse := (old_jts[j+1] - move_3dof[j+1])**2;
           sum := mse + sum;

           jt_str[j]:=word_to_string(make_word(roundoff(move_3dof[j+1]*1000)));
        end;

        old_jts := move_3dof;

        if (sum > (1.0) ) then
           begin
              str_send := curr_rob*' '*jt_str[0]*' '*jt_str[1]*' '*jt_str[2];
              write_msg('SENDING JOINT VECTOR THROUGH SOCKET');
              sil_sock1_data(file_d,str_send);
           end
        else
           writeln('NOT ENOUGH CHANGE IN JTS TO SEND THROUGH SOCKET');

     sum := 0.0;
     end;

     if ((curr_rob = 'PR') or (curr_rob = 'PL')) then
     begin
       jt_str := array_create(string,5);
        move_6dof := c_jv(curr_rob);

        for j := 0 to 5 do
        begin
           mse := (old_jts[j+1] - move_6dof[j+1])**2;
           sum := mse + sum;

           jt_str[j]:=word_to_string(make_word(roundoff(move_6dof[j+1]*1000)));
        end;

        old_jts := move_6dof;

        if (sum > (6.0) ) then
           begin
              str_send := curr_rob*' '*jt_str[0]*' '*jt_str[1]*' '*jt_str[2]*'
                          '*jt_str[3]*' '*jt_str[4]*' '*jt_str[5];
```

```
                write_msg('SENDING JOINT VECTOR THROUGH SOCKET');
                sil_sock1_data(file_d,str_send);
             end
          else
             writeln('NOT ENOUGH CHANGE IN JTS TO SEND THROUGH SOCKET');

       sum := 0.0;
       end;

       if ((curr_rob = 'FR') or (curr_rob = 'FL')) then
       begin
         jt_str := array_create(string,8);
          move_9dof := c_jv(curr_rob);

          for j := 0 to 8 do
          begin
             mse := (old_jts[j+1] - move_9dof[j+1])**2;
             sum := mse + sum;

             jt_str[j]:=word_to_string(make_word(roundoff(move_9dof[j+1]*1000)));
          end;

          old_jts := move_9dof;

          if (sum > (9.0) ) then
             begin
                str_send := curr_rob*' '*jt_str[0]*' '*jt_str[1]*' '*jt_str[2]*'
                            '*jt_str[3]*' '*jt_str[4]*' '*jt_str[5]*'
                            '*jt_str[6]*' '*jt_str[7]*' '*jt_str[8];
                write_msg('SENDING JOINT VECTOR THROUGH SOCKET');
                sil_sock1_data(file_d,str_send);
             end
          else
             writeln('NOT ENOUGH CHANGE IN JTS TO SEND THROUGH SOCKET');

       sum := 0.0;
       end;

end;

{MAKE THE PREVIOUS PROCEDURE A CLOSURE FOR LATER USE}
print_cjv_msg_cl == mk_closure("print_cjv_msg,map(ob));
{----------------------------- print_cjv_msg CLOSURE ----------------------}


process cs_move(joint_file:string;rob:string);
begin
     wait(joint_file);
     wait(rob);
     moveto_tabjv(rob,joint_file);
end;
```

```
process show(numrobs:integer);
var
     which_rob : anno_string;
     rob_cs    : string;
     rob_msg   : string;
     filename  : string;
begin
     wait(numrobs);

   for i := 1 to numrobs do
   begin
     if (curr_cell = '2x9') then
        begin
           write_msg('SELECT A ROBOT FOR MOTION : [FL, FR]:');
           which_rob := get_a_string();
           if ((which_rob.val <> 'FL') and (which_rob.val <> 'FR')) then
              begin
                 activate(testbed_menu);
                 write_msg('ROBOT SELECTION NOT VALID - TRY AGAIN');
              end
           else
              begin
                 rob_cs := which_rob.val;
                 if (rob_cs = 'FL') then rob_msg := 'FULL LEFT ARM';
                 if (rob_cs = 'FR') then rob_msg := 'FULL RIGHT ARM';
              end;
        end;

     if (curr_cell = '2x6+2x3') then
        begin
           write_msg('SELECT A ROBOT FOR MOTION: [PL, PR, CL, CR]:');
           which_rob := get_a_string();
           if ((which_rob.val <> 'PL') and (which_rob.val <> 'PR') and
               (which_rob.val <> 'CL') and (which_rob.val <> 'CR')) then
              begin
                 activate(testbed_menu);
                 write_msg('ROBOT SELECTION NOT VALID - TRY AGAIN');
              end
           else
              begin
                 rob_cs := which_rob.val;
                 if (rob_cs = 'PL') then rob_msg := 'LEFT PUMA';
                 if (rob_cs = 'PR') then rob_msg := 'RIGHT PUMA';
                 if (rob_cs = 'CL') then rob_msg := 'LEFT PLATFORM';
                 if (rob_cs = 'CR') then rob_msg := 'RIGHT PLATFORM';
              end;
        end;

        start(cs_move);
        write_msg(concat('ENTER THE path/filename WHICH CONTAINS JOINT VECTORS
                         FOR THE ',rob_msg));
        filename := new_read();
```

```
            signal(cs_move,joint_file,filename);
            signal(cs_move,rob,rob_cs);
         end;
    end;

    process show_1();
    begin
         signal(show,numrobs,1); end;

    process show_2();
    begin
         signal(show,numrobs,2); end;

    process show_3();
    begin
         signal(show,numrobs,3); end;

    process show_4();
    begin
         signal(show,numrobs,4); end;

    {------------------------------ show PROCESS ------------------------------}


    {process moving_9dof(rob_name:string;joint_vec:darray of real);
    var
         temp_vec : jv;
    begin
         wait(rob_name);
         wait(joint_vec);

            end3[0]:=joint_vec[6]; end3[1]:=joint_vec[7]; end3[2]:=joint_vec[8];
           temp_vec[]:=joint_vec[];temp_vec[]:=joint_vec[];temp_vec[]:=joint_vec[];
           temp_vec[]:=joint_vec[];temp_vec[]:=joint_vec[];temp_vec[]:=joint_vec[];
            temp_vec[6]:=end3;
         moveto(rob_name,temp_vec);
    end;

    process moving_6dof(rob_name:string;joint_vec:jv);
    begin
         wait(rob_name);
         wait(joint_vec);
         moveto(rob_name,joint_vec);
    end;
    v
    process moving_3dof(rob_name:string;joint_vec:darray of real);
    var
         temp_vec :
    begin
         wait(rob_name);
         wait(joint_vec);
         moveto(rob_name,joint_vec);end;
```

```
}


{===== MENU HANDLER PROCEDURES =====}
{---------- TOP MENU ----------}

procedure testbed_menu_handler(s:string);
var
     sure : anno_string;
begin
     case s of
{        '!INIT!'        : }
        'SOCKET CONNECT': begin
           if (file_d <> 0) then
              write_msg('SOCKET SERVER IS NOT READY FOR CONNECTION')
           else
              call(connect_menu);
           end;
        'CELL'           : call(cell_menu);
        'CURR ROBOT'     : begin
           current_rob();
           set_msg(testbed_menu,'CURR ROBOT',curr_rob);
           end;
        'HOME'           : call(home_menu);
        'MOTION'         : call(motion_menu);
        'OFF'            : sendingOff();
        'GRIPPER'        : begin
           if (curr_rob = 'none') then
              write_msg('SELECT ROBOT BEFORE ACTUATING GRIPPER FUNCTIONS')
           else
              call(gripper_menu);
           end;
        'COMPLIANCE'     : begin
           if (curr_rob = 'none') then
              write_msg('SELECT ROBOT BEFORE ENGAGING COMPLIANCE FUNCTION')
           else
              if ((curr_rob = 'CL') or (curr_rob = 'CR')) then
                 write_msg('NO COMPLIANCE OPTION FOR CURRENT ROBOT')
              else  begin
                 comp_o_f();
                 set_msg(testbed_menu,'COMPLIANCE',comp);
              end;
           end;
        'CURRENT POSITION': currentPosition();
        'END SESSION'    :
           begin
              write_msg('WILL TERMINATE LINK TO CTOS, REQUIRING REBOOT -
                         SURE? (y|n)');
              sure := get_a_string();
              if (((sure.val <> 'y') and (sure.val <> 'n')) or
                                                   (sure.val = 'n')) then
                 write_msg('LINK TO CTOS NOT TERMINATED');
```

```
                if (sure.val = 'y') then
                begin
                   write_msg('SENDING MESSAGE TO QUIT');
                   sil_sock1_data(file_d,'q');
                   exit_user_tree();
                   write_msg('THANK YOU FOR USING THE CIMSTATION/CTOS GRAPHICAL
                              INTERFACE');
                end;
             end;

          '!ABORT!' : exit_user_tree();
      end;
end;

{----- SUB MENUS -----}

procedure connect_menu_handler(s:string);
var
     hn_anno : anno_string;
     pn_anno : anno_real;
begin
     case s of
{         '!INIT!' : }
         'HOSTNAME'       : begin
            write_msg('ENTER MACHINE NAME THAT WILL ACT AS HOST TO SOCKETS');
            hn_anno := get_a_string();
            host_name := hn_anno.val;
            set_msg(connect_menu,'HOSTNAME',host_name);
            end;
         'PORTNUMBER'     : begin
            write_msg('ENTER PORT NUMBER SOCKET CONNECTION (INTEGER)');
            pn_anno := get_a_real();
            port_n := roundoff(pn_anno.val);
            set_msg(connect_menu,'PORTNUMBER',port_n);
            end;
         'CONNECT - OK' : begin
            writeln('SHOULD BE CONNECTING');
            file_d := sil_call1_client(host_name, port_n);
                                      { OPEN SOCKET CONNECTION W/CTOS }
            writeln(file_d);
            activate(testbed_menu);
            end;
         '!ABORT!' : menu_return();
     end;
end;  {---------- END connect_menu_handler ----------}

procedure cell_menu_handler(s:string);
begin
     case s of
        '!INIT!' :
           write_msg('WHICH TESTBED FORMAT WILL YOU BE USING?');
        '!ABORT!': menu_return();
```

```
        '2x9DOF ARMS'  : begin
           curr_cell := '2x9';
           run(cell_2bots);
           activate(testbed_menu);
           write_msg('THIS IS THE CIRSSE TESTBED');
           end;

        '2x6DOF+2x3DOF ARMS': begin
           curr_cell := '2x6+2x3';
           run(cell_4bots);
           activate(testbed_menu);
           write_msg('THIS IS THE CIRSSE TESTBED');
           end;
     end;
end;  {---------- END cell_menu_handler ----------}


procedure home_menu_handler(s:string);
var
     where : string;
begin
     case s of
{        '!INIT!'  : }
        '!ABORT!' : menu_return();

        'CURRENT TO HOME' : begin
           run(send_one_to_r,one_to);
           write_msg(concat(curr_rob_msg,' IS AT THE CIRSSE HOME POSITION'));
           end;
        'ALL TO HOME'     : begin
           run(send_all_to_r,all_to);
           write_msg('ALL ROBOTS ARE AT THE CIRSSE HOME POSITION');
           end;
        'CURRENT TO ZERO'  : begin
           run(send_one_to_z,one_to);
           write_msg(concat(curr_rob_msg,' IS AT THE ZERO POSITION'));
           end;
        'ALL TO ZERO'      : begin
           run(send_all_to_z,all_to);
           write_msg('ALL ROBOTS ARE AT THE ZERO POSITION');
           end;
     end;
end;  {---------- END home_menu_handler ----------}


procedure motion_menu_handler(s:string);
var
     ok             : anno_string;
     numrobs, k     : integer;
     numrobs_string: anno_string;
     refr_freq      : anno_real;
```

```
begin
     case s of
        '!INIT!' : write_msg('SELECT ROBOT(S) MOTION TYPE ----->
                          THEN MOVE ROBOT(S) IN DESIRED MANNER');
        '!ABORT!' : menu_return();

        'TO GOAL' : begin
           refresh(0.0);
           write_msg('GOAL MOTION DEPENDS %100 ON THE CTOS TG -
                      ONLY GOAL POSITIONS SENT');
           end;

        'VIA PATH' : begin
           write_msg('VIA PATH MOTION USES CS TG - SELECT FREQ OF JT ANGLE
                      NOTIFICATION[sec]');
           refr_freq := get_a_real();
           refresh(refr_freq.val);
           end;

        'PREVIEW/CREATE' :  begin
           exit_user_tree();
           write_msg('USE CS FUNCTIONS TO CREATE/VIEW DESIRED PATH, THEN
                      -> "~/MOVE/ON".');
           end;

        'VIEW STORED PATH' :  begin
           if ((curr_cell <> '2x9') and (curr_cell <> '2x6+2x3')) then
           begin
              write_msg('CHOOSE A CELL BEFORE TRYING TO MOVE ROBOTS');
              menu_return();
           end
           else
           begin
              write_msg('ENTER NUMBER OF COMPONENT ROBOTS INVOLVED IN THE PATH
                         (1-4)');
              numrobs_string := get_a_string();
              if numrobs_string.val = '1' then run(show_1,show,cs_move);
              if numrobs_string.val = '2' then run(show_2,show,cs_move);
              if numrobs_string.val = '3' then run(show_3,show,cs_move);
              if numrobs_string.val = '4' then run(show_4,show,cs_move);
              menu_return();
           end;

           end;

        'ON'     : begin
           write_msg('ON = TESTBED ARMS WILL RESPOND TO NEXT GRAPHICS MOVE,
                      OK? (y|n)');
           ok := get_a_string();
           if ((ok.val <> 'y') and (ok.val <> 'Y') and
                          (ok.val <> 'n') and (ok.val <> 'N')) then
           begin
```

```
            activate(motion_menu);
            write_msg('CHOOSE AGAIN: MUST ENTER y|n TO ENGAGE TESTBED.');
         end;

         if ((ok.val = 'n') or (ok.val = 'N')) then menu_return();

         if ((ok.val = 'y') or (ok.val = 'Y')) then
         begin
            {CHECK THAT A ROBOT IS SELECTED}
            if (curr_rob <> 'none') then
            begin

            refresh_actions := cons(print_cjv_msg_cl,refresh_actions);

            exit_user_tree();
            write_msg(concat(curr_rob_msg,' ENGAGED FOR MOTION - USE CS
                     FUNCTIONS, TO START TESTBED'));
            end;
         end;

         end;     {----- ON -----}

      'OFF'    : begin
         refresh_actions := cdr(refresh_actions);  {RESETS REFRESH_ACTIONS}
         old_jts := old_jts_init;

         activate(testbed_menu);
         write_msg('OFF = TESTBED ARMS NO LONGER RESPONDING TO CIMSTATION
                        MOVES.');
         end;     {----- OFF -----}

    end;  {----- case -----}
end;     {---------- END motion_menu_handler ----------}


procedure gripper_menu_handler(s:string);
begin
    case s of
      '!INIT!'  : set_msg(gripper_menu,'CURR ROBOT',curr_rob);
      'OPEN'    : begin
         if ((curr_rob = 'CL') or (curr_rob = 'CR')) then
         begin
            write_msg('THIS ROBOT DOES NOT HAVE A GRIPPER');
            activate(testbed_menu);
         end

         else begin   {CASE OF ROBOT WITH GRIPPER}
            if ((grip = 'not set') or (grip = 'close')) then begin
               run(gripper_o);
               grip := 'open';
               write_msg('GRIPPER OF **CURRENT** ROBOT IS NOW OPEN');
               sil_sock1_data(file_d,('G O '*curr_rob));
```

```
                    writeln('GRIPPER OPEN SENT');
                 end
                 else
                    write_msg('GRIPPER IS ALREADY OPEN');
                 end;
              end;

           'CLOSE'   : begin
              if ((curr_rob = 'CL') or (curr_rob = 'CR')) then
              begin
                 write_msg('THIS ROBOT DOES NOT HAVE A GRIPPER');
                 activate(testbed_menu);
              end

              else begin   {CASE OF ROBOT WITH GRIPPER}
                 if ((grip = 'not set') or (grip = 'open')) then begin
                    run(gripper_c);
                    grip := 'close';
                    write_msg('GRIPPER OF **CURRENT** ROBOT IS NOW CLOSED');
                    sil_sock1_data(file_d,('G C '*curr_rob));
                    writeln('GRIPPER CLOSE SENT');
                 end
                 else
                    write_msg('GRIPPER IS ALREADY CLOSED');
                 end;
              end;

           '!ABORT!' : activate(testbed_menu);

        end;  {----- case -----}
     end;     {---------- END gripper_menu_handler ----------}

     {===== testbed_menu IS TOP LEVEL USER MENU =====}

     top_user_menu := testbed_menu;
```

```
{
  Package: Cimstation Socket Interface
  File: sil_sock.sil
  Written by: Anna Hron, Steve Murphy
  Date: 19 Oct 92
  Purpose: To connect to sockets and output joint angles.
  Modification History:
     NOW ACCEPTS VARIABLE HOSTNAME AND PORTNUMBER IN CALL_CLIENT ROUTINE
                                                   - ABH - 6 NOV 92
}

import("callclient,map(integer,ob,integer));

function sil_call1_client(hn_string:string; p_num:integer):integer;
begin
  sil_call1_client:=callclient(hn_string as_type ob, p_num);
end;

import("callsockdata,map(integer,integer,ob));

function sil_sock1_data(fd_server:integer; sil_string: string): integer;
begin
  sil_sock1_data:=callsockdata(fd_server,sil_string as_type ob);
end;


import("callsockrecv,map(integer,integer,ob));

function sil_sock_recv(fd_server:integer; rec_string:string):integer;
begin
   sil_sock_recv:=callsockrecv(fd_server,rec_string as_type ob);
end;
```

```
/* THIS CODE WILL CREATE A SOCKET AND SEND A MESSAGE.
   THE SUPPORTING CODE IS ALSO INCLUDED.
   THIS IS CODE FROM /home/nicewarn/Editing/sockLib.

   ABH - 14 OCT 92

   MODIFICATIONS : call_client NOW ACCEPTS VARIABLES FOR HOSTNAME
                   AND PORTNUMBER
*/


#include <stdio.h>
#include <stdlib.h>
#include <lisp.h>  /*  RECOMMENDED INCLUDES - STEVE M. */
#include <compiled.h>
#include <precomp.h>
#include <postcomp.h>

#include <unistd.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>

/* #include "sockLib.h" */

#ifndef INCsockLibH
#define INCsockLibH

#define MAX_HOST_NAME_LENGTH   128


typedef int SOCKET_TYPE;

#endif INCsockLibH


/* DUMMY FUNCTION REQUIRED BY SIL */
init_c_sock() {}

/* MAIN PROGRAM WHICH ACCEPTS "sockClient(<hostname> <portnum>)"
   IT ESTABLISHES A CONNECTION WITH THE SOCKET RUNNING IN CTOS
*/
int callclient(hnstring, pnum)
stringob hnstring;
int pnum;
{
```

```
   char *hostname;

   /* CONVERT hnstring (SIL) TO hostname (C) */
   hostname = l2c_str(hnstring);

   return(sockClient(hostname, pnum));
}


int sockClient(hostname, portnum)
char hostname[128];
unsigned short portnum;
{
   int fdServer;
   struct sockaddr_in addrServer;

  if((fdServer = sockConnect(hostname, portnum, &addrServer, SOCK_STREAM)) < 0)
      return(-1);
   printf("Got stream server socket.\n");
   return(fdServer);
}

int callsockdata(fdServer, silString)
int fdServer;
stringob silString;
{
  char *buf;

  /* Convert Silstring to C string */
  buf = l2c_str(silString);

  /* send data to the socket */
  return(sockData(fdServer, buf));
}


int callsockrecv(fdServer, recString)
int fdServer;
stringob recString;
{
   char *buf;

   /* CONVERT SILSTRING TO C STRING */
   buf = l2c_str(recString);

   /* READ DATA FROM THE SOCKET */
   return(sockRecv(fdServer,buf));
}


/* SENDS DATA TO SOCKET RUNNING IN CTOS */
```

```
int sockData(fdServer, buf)
int fdServer;
char buf[350];
{
   if(sockStrmSend(fdServer, buf, 350) < 0)
      return(-1);
   return(1);
}


/* READS DATA FROM SOCKET SERVED BY CTOS */

int sockRecv(fdServer,buf)
int fdServer;
char buf[350];
{
   if (sockStrmRecv(fdServer, buf, 350) < 0)
      return(-1);
   return(1);
}


/* connect to an existing socket on a given port on a given host */
int sockConnect(hostname, portnum,
sa, sockType)
char *hostname;
unsigned short portnum;
struct sockaddr_in *sa;
SOCKET_TYPE sockType;
{
   struct hostent *hp;
   int s;

   if((hp = gethostbyname(hostname)) == NULL)
   {
      errno = ECONNREFUSED;
      printf("Unknown host '%s'.\n", hostname);
      return(-1);
   }

   bzero(sa, sizeof(struct sockaddr_in));
   bcopy(hp->h_addr, (char *)&sa->sin_addr, hp->h_length);
   sa->sin_family = hp->h_addrtype;
   sa->sin_port = htons((u_short)portnum);

   if((s = socket(hp->h_addrtype, sockType, 0)) < 0)
   {
      perror("Error getting server stream socket");
      return(-1);
   }
   if(connect(s, (struct sockaddr *)sa, sizeof(struct sockaddr_in)) < 0)
   {
```

```
      close(s);
      perror("Error connecting to server stream socket");
      return(-1);
   }
   return(s);
}


/*
** Stream socket I/O routines
*/

/* write data to stream socket */
int sockStrmSend(s, data, dataSize)
int s;
register char *data;
unsigned dataSize;
{
   register int bcount = 0; /* counts bytes written */
   register int br;         /* bytes written this pass */

   while(bcount < dataSize)
   {
      if((br = write(s, data, dataSize - bcount)) > 0)
      {
         bcount += br;
         data += br;
      }
      else if(br < 0)
      {
         perror("sockStrmSend");
         return(br);
      }
   }
   return(bcount);
}


/* read data from stream socket */
int sockStrmRecv(s, data, dataSize)
int s;
register char *data;
unsigned dataSize;
{
   register int bcount = 0; /* counts bytes read */
   register int br;         /* bytes read this pass */

   while(bcount < dataSize)
   {
      if((br = read(s, data, dataSize - bcount)) > 0)
      {
         bcount += br;
```

```
 data += br;
      }
      else if(br < 0)
      {
 perror("sockStrmRecv");
 return(br);
      }
   }
   return(bcount);
}
```